NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THREE-DIMENSIONAL FRACTAL MOUNTAINS

by

Patricia J. Collins

Thesis Advisor: Michael J. Zyda

Co-Advisor: Harold M. Fredricksen

Approved for public release; distribution is unlimited

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION / DOWNGRADING SCHEDULE	
ERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION val Postgraduate School	6b OFFICE SYMBOL (If applicable) Code 53
	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (City, State, and ZIP Code) nterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey,CA 93943-5000
NAME OF FUNDING SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

TITLE (Include Security Classification)

HREE-DIMENSIONAL FRACTAL MOUNTAINS

PERSONAL AUTHOR(S)
ollins, Patricia J.

TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
ister's Thesis	FROM _____ TO _____	December 1988	164

SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and o not reflect the official policy or position of the Department of Defense or he U.S. Government

COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Fractals, three-dimensional mountains, midpoint displacement technique

ABSTRACT (Continue on reverse if necessary and identify by block number)

his study provides a guide to a series of systematic techniques used to create ractal mountains. The fractal mountains are created through an Interactive ystem for Fractal Mountains (ISFM). To create the fractal mountains in ISFM modified midpoint displacement technique in three dimensions is used. ugmenting the midpoint displacement algorithm is a random number generator, hat provides randomness in the displacement so as to simulate nature. These wo algorithms plus an algorithm for lighting and for shading allow the user to evelop different types of fractal mountains. When creating a fractal mountain ith ISFM the user has the options of placing the location of the light source or the time of day, of determining the ruggedness or texture of the mountain nd of positioning the outline for a mountain range. ISFM generates a fractal ountain or a fractal mountain range on an IRIS workstation. ISFM provides a ystematic and tutorial approach to creating fractal mountains that can be asily repeated by others.

DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
a NAME OF RESPONSIBLE INDIVIDUAL ofessor Hal Fredricksen	22b TELEPHONE (Include Area Code) (408) 646-2206
	22c OFFICE SYMBOL Code 53

D FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office 1986—606-24.

Approved for public release; distribution is unlimited

Three-Dimensional Fractal Mountains

by

Patricia J. Collins
Civilian, Naval Postgraduate School
B.S. University of California at Santa Barbara, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

ABSTRACT

This study provides a guide to a series of systematic techniques used to create fractal mountains. The fractal mountains are created through an Interactive System for Fractal Mountains (ISFM). To create the fractal mountains in ISFM a modified midpoint displacement technique in three dimensions is used. Augmenting the midpoint displacement algorithm is a random number generator that provides randomness in the displacement so as to simulate nature. These two algorithms plus an algorithm for lighting and for shading allow the user to develop different types of fractal mountains. When creating a fractal mountain with ISFM, the user has the options of placing the location of the light source for the time of day, of determining the ruggedness or texture of the mountain and of positioning the outline for a mountain range. ISFM generates a fractal mountain or a fractal mountain range on an IRIS workstation. ISFM provides a systematic and tutorial approach to creating fractal mountains that can be easily repeated by others.

TABLE OF CONTENTS

I. INTRODUCTION

A. FRACTAL GEOMETRY AS A DESCRIPTION OF NATURE

Fractal Geometry originated in the late 1970's with Benoit Mandelbrot [ref. 1]. Mandelbrot felt that lines, circles and spheres of Euclidean geometry were unable to describe such patterns of nature as clouds, coastlines, trees and mountains. He developed the concept of fractal geometry to describe these patterns of nature. Although the concept of Fractal Geometry was a new concept it includes earlier work of Peano, Cantor and Koch [ref. 1]. Peano's space filling curves, Cantor's dust concept and the Koch curves have all been incorporated into the theory of fractal geometry. A Peano curve is a mapping of a two dimensional curve on the plane (2D) which completely fills the two dimensional space. The Cantor dust concept, also called Cantor's middle thirds set is a topological set of dimension 0. The set is created on the closed interval by dividing the interval [0,1] into three pieces and then removing the middle third. Each of the remaining pieces are further divided and then the middle thirds removed in the same way and so on to infinity. Koch curves are particularly relevant to this study because the Koch snowflake is the first application of the midpoint displacement technique used here to generate fractal mountains [ref. 1, pp. 1-4].

B. KOCH SNOWFLAKE

The Koch snowflake is one of the most well known fractal curves. The generation of the snowflake illustrates part of the approach taken in this study. The snowflake is created by starting with a basic Euclidean shape. A line segment, such as AB in Figure 1.1a, is used as an initiator. The midpoint of the initiator (or base line segment), C', is displaced to a point C as in Figure 1.1b. The initiator AB is replaced by the pair of lines AC and CB. The midpoints of line segments AC and CB are then displaced back to the original line at points D and F respectively. By connecting D to C by the line DC and C to F by the line CF the resulting polygonal line, ADCFB shown in Figure 1.1c, is called the generator. The process used to create the generator is called the midpoint displacement technique.

To create the fractal or Koch snowflake, the process of creating generators out of each initiator is performed recursively. A process or a function is said to be recursive if, in the course of its execution, the function issues a call to itself [ref. 2, pp. 37-38]. In the case of the Koch snowflake, as in any recursive process, creating generators from each new initiator is, in theory, an infinite process. However, repeating a process or function infinitely often in this way is not possible, except theoretically. On finite equipment such as a computer there must be a criterion

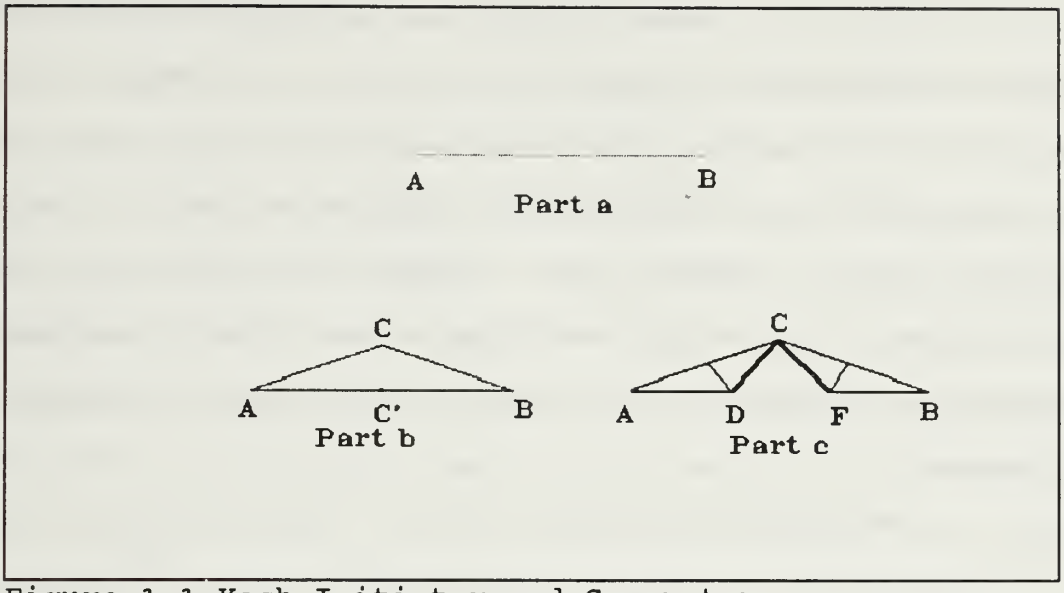

Figure 1.1 Koch Initiator and Generator

developed to stop the process. The criterion for stopping is developed with the limitations of the computer being used in the recursion kept in mind. For example, some smallest line segment length may be incorporated into a stopping criterion.

C. THREE DIMENSIONAL EXTENSION

In this study, the midpoint technique has been extended to apply to three dimensional objects. To extend the midpoint displacement technique from two to three dimensions, three line segments in the form of a triangle are used as initiators. This triangle, on which the fractal mountain is created, is called a base triangle and, as can be seen in Figure 1.2, is placed in the xz plane. The right handed coordinate system shown in Figure 1.2 is the coordinate system

that is used in this study. The y coordinate is viewed as the vertical direction and negative values of z are taken as going into the plane as the viewer perceives the vectors in the coordinate system. The base triangle can be taken to be any shape or size. By the use of many base triangles, the development of a fractal mountain scene rather than just a single mountain as in the case of one base triangle is possible.

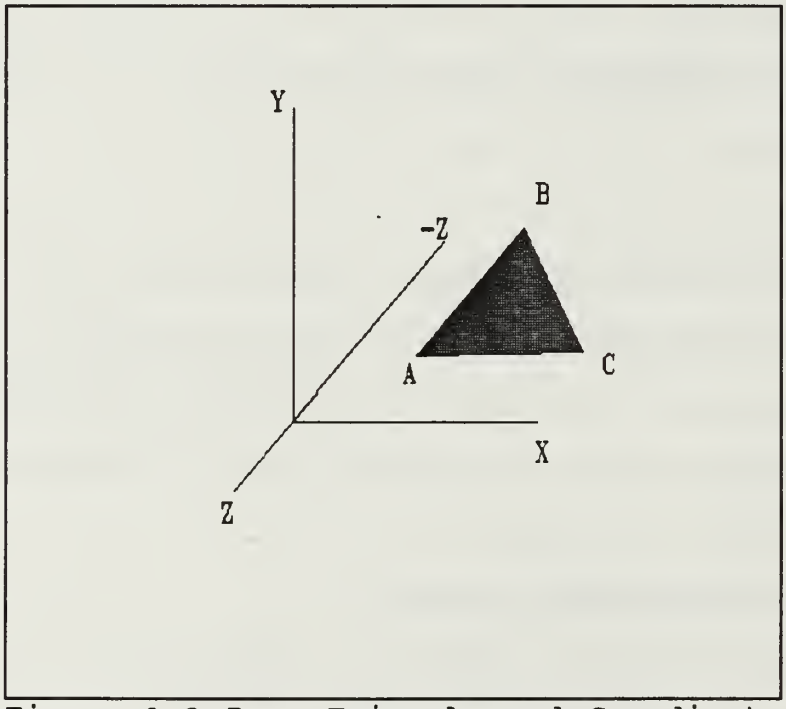

Figure 1.2 Base Triangle and Coordinate System

D. MOTIVATION

Most of the earlier literature on fractal mountains has been developed from an artistic approach. The artistic

approach gives the desired mountain views, but it is difficult for someone other than the artist to repeat the results as no systematic approach is available. More recent literature has included some algorithms on the different techniques for creating fractal mountains. In order to formalize the tinkering or artistic tricks used in the creation of a realistic picture, fractal mountains must be developed using interactive algorithms that can be easily understood and duplicated by others. Further improvements may then be added to enhance and expand the possible types of fractal mountains.

E. SUMMARY OF THE REST OF THE STUDY

In Chapter II, a literature review of the historical development of fractal mountains is given. The basic theory and algorithms used to create the fractals are discussed and a stopping criterion based on machine limitations on line segment length also appears there. In Chapter III, the main technical limitations to the quality of fractal mountain creation, granularity and gaps, are discussed. In Chapter IV, the problems connected with algorithms used in solving the granularity and gapping problems are discussed. In Chapter V, the final implementation used in this study to achieve a natural look in the fractal mountain generation is given. The final chapters explain the Interactive System for Fractal Mountains (ISFM). ISFM is implemented by pop-up menus which provide the options needed to change the texture of the

fractal mountain and to change the light source position for either a mountain or a mountain range. The contour for the mountain range can also be defined interactively. The description of ISFM is followed by actual pictures showing fractal mountains.

II. FRACTAL MOUNTAINS

A. HISTORIC BACKGROUND

The application of fractals has become more extensive since the fractal concept was originated in the late 1970's. Mort La Brecque [ref. 3] cites applications ranging from chemical reactions, protein behavior, mechanical and electrical system behavior to natural objects. The natural objects created by fractals include galactic clusters, earthquake patterns, rainfall, clouds and landscapes. The results obtained in each of these applications have been enhanced by advances achieved in computer graphics. With the continued expanding technology of the computer, images created by fractals help to illustrate and explain many different aspects of all the applications discussed.

Fractals were first developed using the ideas of self-similarity [ref. 1,4,5,6] and self-affinity [ref. 5,6]. Michael F. Goodchild [ref. 4] defines self-similarity as a property of certain curves whereby each part of the curve is indistinguishable from the whole curve and by which, on each different level of scale, the finer structure of the curve resembles the grosser structure. Self-affinity is defined by Richard F. Voss. [ref. 5] Consider a set S = {x,y,...} of points defined, as $x=(x_1,....,x_E)$, in a Euclidean space of dimension E. Under an affine transformation, each of the E

coordinates of x may be scaled by a different ratio r = $(r_1, r_2, \ldots . r_E)$. The set S is transformed to rS with points at $r(x) = (r_1x_1, \ldots, r_Ex_E)$. A bounded set S is called self-affine when S is the union of N distinct (non-overlapping) subsets each of which is congruent to rS. Self-similarity and self-affinity are essentially geometric scaling properties. The Koch snowflake discussed in Chapter I is an example of a curve that has self-similarity. That is, as we iteratively produce the snowflake by repeated operations of the generator on each new initiator for each level of scale, the same structure for each level is repeated yet finer.

There are several other effective fractal methods which can be used to model natural objects. Demko, Hodges and Naylor [ref. 7] produce iterated function systems for generating fractals in two dimensions(2D). These iterated systems can produce many of the classic 2D deterministic fractals found in Mandelbrot [ref. 1]. Another fractal method used to represent nature is based on the recursive subdivision method [ref. 8,9,10,11,12]. Amburn, Grant and Whitted [ref. 8] indicate that the recursive subdivision procedure has three components, namely:

1) framework
2) subdivision equation
3) termination criterion

The framework is the overall organization of the procedure and controls the order of surface subdivisions. The subdivision equations describe the shape of the original

polygon and the shapes of the iterated polygons at each level. The termination criterion is used in ending the fractal subdivision process. Some of the subdivision equations described by Gavin Miller [ref. 9] can result in triangle-edge subdivisions, diamond-square subdivisions or square-square subdivisions. Loren C. Carpenter [ref. 10] and Alan Ray Smith [ref. 11] describe yet another subdivision equation based on a triangular subdivision. The termination criterion is affected by the computer that is being used and is employed to stop the recursion process at the level in which the criterion is met.

Carpenter [ref. 12] describes an additional subdivision method which divides a fractal region of finite size in such a way as to maintain self-similarity in the object created and thereby the statistical properties of the object. Carpenter's method determines the midpoint of a fractal curve that is to be subdivided by a constrained random process. Fournier, Fussell and Carpenter [ref. 13] give algorithms for subdividing scalar displacements of one dimensional noise and two dimensional polygons. The subdivision of a fractal curve at its midpoint is also known as the midpoint displacement technique [ref. 1, pp. 233-234].

The actual first use of the midpoint displacement technique was for creating the Koch snowflake. The Koch snowflake uses an initiator and a generator [ref. 1, pp. 39-43] and replaces each initiator by the generator during each

step in the recursive process. The displacement of the midpoint for the Koch snowflake was predetermined so that the generator had a specified shape. The method of displacing the midpoint is the most critical aspect of the midpoint displacement technique. There are three variations suggested for the midpoint displacement technique described in [ref. 1, pp. 233-234]. They are the transversal midpoint displacement, the isotropic directions displacement and the random lengths displacement. The transversal midpoint displacement method allows the displacement of the midpoint to the left or to the right along the initiator according to specific rules as one moves away from the center. The isotropic displacement directions method randomizes the displacement directions of the midpoint. That is, if the line lies along the x axis, the displacement direction would be either positive y (up) or negative y (down) i.e., orthogonal to the x axis and to the z axis. The random displacement lengths method allows the length or magnitude of the displacement to be random. Thus, the distance from the midpoint to the actual displaced point varies randomly. Using both the isotropic displacement directions method and the random displacement lengths method together affords complete randomness in the use of the midpoint displacement technique and is termed the random displacement method.

The random displacement method, as described by Peitgen [ref. 14, pp. 133-136], is produced by the use of two

components. These are random variables and random functions. The random variables are used in the random functions and are created by a random number generator. The random function resulting from the random variables that are used, as Peter Sorensen [ref. 15, p. 160] indicates, achieve a displacement in the y direction. The y displacements are either up or down in the screen display, denoting positive or negative displacement, respectively, and are thus applicable in the creation of the surface of fractal mountains. When creating fractal mountains, the y direction displacements yield the altitudes of the various peaks and valleys. Care must be employed in the generation of the random numbers to prevent the random variables from becoming completely chaotic.

Richard Voss [ref. 5] states that the random displacement method is a recursive generating technique first applied to the study of normal Brownian motion as early as the 1920's by N.Wiener. In Voss's explanation, the midpoint values for the random midpoint displacements are determined by the equations:

$$V(+1/2) = 0.5(V(0) + V(1)) + \blacktriangle_1 \quad \text{(equation 2.1)}$$

$$V(-1/2) = 0.5(V(0) + V(-1)) + \blacktriangle_2 \quad \text{(equation 2.2)}$$

where V(0) and V(1) and V(0) and V(-1) are the endpoints of the two initiators and $\blacktriangle_1$ and $\blacktriangle_2$ are gaussian random variables with zero mean and standard deviation 1.

Equations 2.1 and 2.2 yield the displaced midpoints of the two line segments V(0)V(1) and V(-1)V(0) as illustrated in Figure 2.1. The initial portion of the equations determines the midpoint of the two line segments. The ▲'s are the random

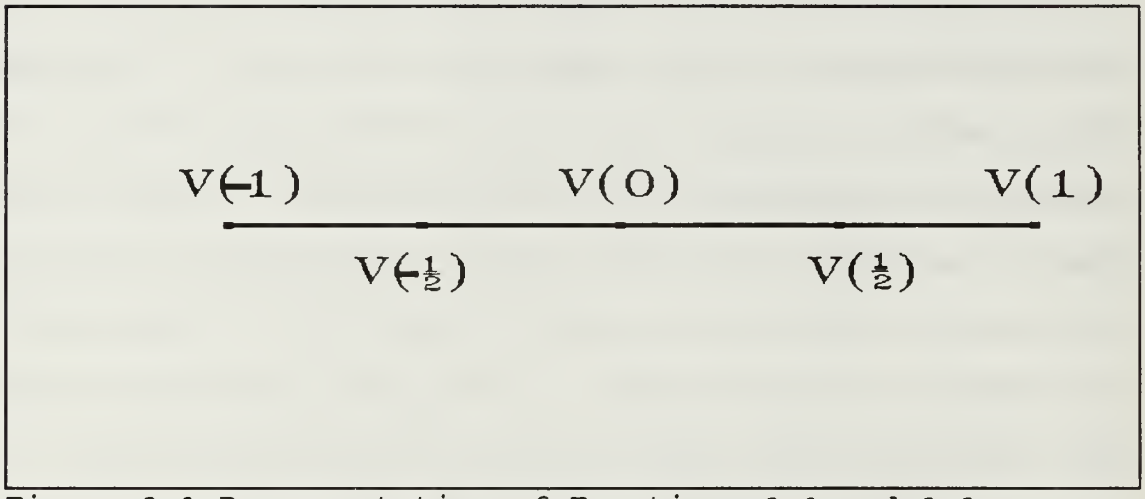

Figure 2.1 Representation of Equations 2.1 and 2.2

variables that determine the size of the displacement of the midpoint.

Equations 2.1 and 2.2 are similar to the equations used in the sample algorithms given by Fournier, Fussell and Carpenter [ref. 13] for the scalar displacement of one dimensional noise and of two dimensional polygons.

The random midpoint displacement technique described above was first developed in one and two dimensions. The technique was expanded to three dimensions by Peitgen [ref. 14, pp. 95-105, 244-254]. Peitgen used the extension to three dimensions

to create three dimensional surfaces of fractal mountains. The fractal surface so created and displayed on a computer screen gives a realistic simulation of three-dimensional mountains as described by Patrick Bass [ref. 16]. A true three dimensional fractal mountain must have x, y and z values given for each location. A simple diagram of connections between the vertices of each small triangle by line segments from one base triangle to adjacent triangles can be easily programmed as was done by Michiel van de Panne [ref. 17]. Alex Pentland [ref. 18, 19] interprets the third dimension as determining roughness or smoothness and isotropic or anisotropic conditions of the surface. The third dimension effect can also be described as modeling the texture of the fractal surface. Pentland [ref. 18] describes two ways that the surface texture is affected in image texture. These are 1) projection foreshortening, a function of the angle between the viewer and the surface normal and 2) perspective texture gradient that is due to the increasing distance between the viewer and the surface.

B. BASIC THEORY

In this section the basic framework, the random midpoint displacement technique, is discussed. The termination criterion is discussed in the second section. The third section explains the three dimensional geometry and the last

section discusses the random variables used to displace the midpoints in the midpoint displacement technique.

1. Midpoint Displacement Technique

We return to the study of the recursive subdivision procedure. The random midpoint displacement technique forms the framework for the analysis. To review, the random midpoint displacement technique begins with a line segment and its two endpoints (also called vertices) oriented in three dimensions. The midpoint of the line segment is determined by adding the two endpoints for each of the x, y and z values separately and dividing by two. The second step is to randomly displace the midpoint in the y direction. The coordinate system shown in Figure 1.2 is used. The y coordinate is thus viewed as defining the height or vertical direction. The resulting new value for y with the previous unchanged values for x and z define the new midpoint. The new midpoint forms two new line segments with the two previous endpoints each of half the length of the previous line segment. These are each taken in turn as line segments for the next recursive call to the midpoint displacement routine.

To develop a subdivision equation in three dimensions, one starts with a triangle. Three midpoints are determined with one on each side of the triangle. As seen in Figure 2.2, one starts with a base triangle given by (P1,P2,P3), with each pair of vertices forming a line segment. From the two

endpoints for each edge, the midpoints M_1, M_2 and M_3 are found. By connecting lines between the midpoints M_1, M_2 and M_3, smaller triangles are created, in this case triangles 1, 2, 3 and 4. These new triangles are then recursively operated on separately by the midpoint displacement technique. Thus recursion is implemented on each of the triangles 1, 2, 3 and 4 and the same subdivision technique is applied to each of the triangles in turn.

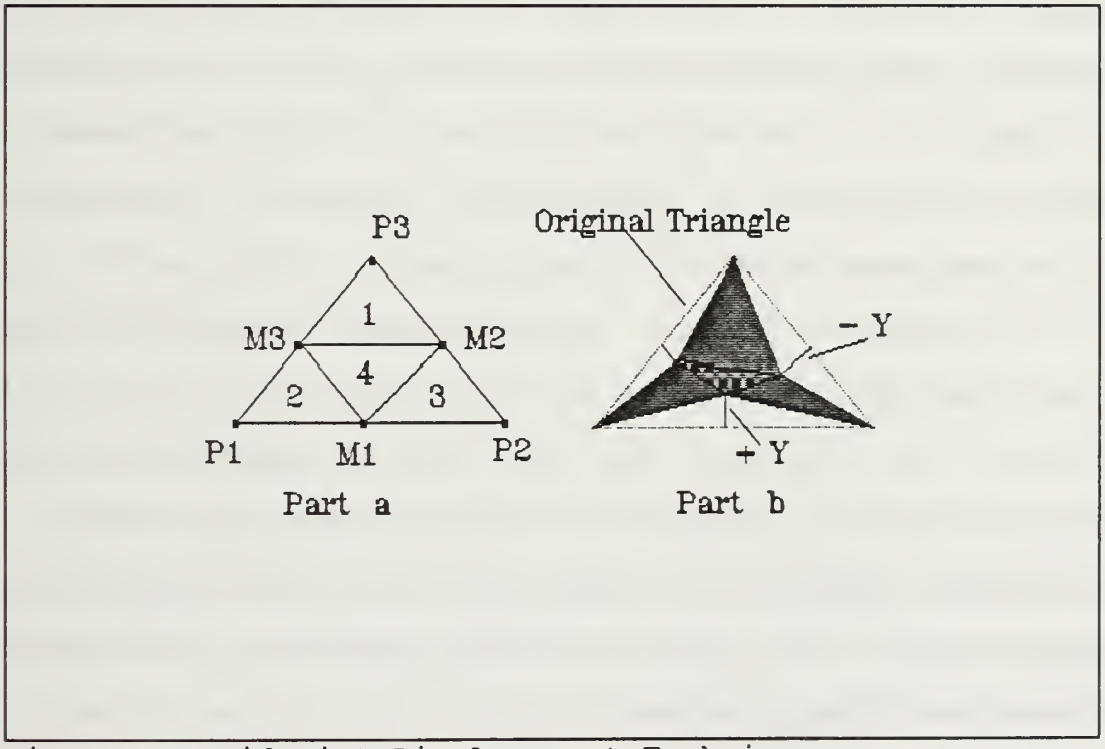

Figure 2.2 Midpoint Displacement Technique

2. Determining the Termination Criterion

The recursive process described in the previous section is continued until some termination criterion is satisfied. The termination criterion is at the control of the programmer and can depend on limitations imposed by the computer used in implementing the recursion. These limitations can also be subject to dynamic memory and disk space, collectively called memory, and screen resolution. Memory requirements imposed by the process can become quite large because each level of recursion creates four triangles from each triangle at the previous level. The more vertices that have to be stored further increase memory requirements. Since there is a limit to system memory, there is also a consequent limit on the number of recursive levels in the process. The amount of memory is known, so one can determine how many triangles and vertices can fit into the amount of space available by a combinatorial analysis. We can also count the numbers of triangles, edges and vertices developed in the recursive process. Each of these classes of objects play a part in determining how many iterations are possible. Each of the triangle, edge and vertex calculations has a separate defining recursion equation for the number of triangles, edges or vertices occurring. The equations are interrelated. For the following equations, the index i denotes the ith iteration.

For each new iteration, the number of triangles is four times the number of the previous iteration. Thus, the recursive equation for the number of triangles (T) created is:

$$T_{i+1} = 4 * T_i \qquad \text{(equation 2.3)}$$

with the initial value $T_0 = 1$

For the number of edges (E) created, the recursive equation is

$$E_{i+1} = 2 * E_i + 3 * T_i \qquad \text{(equation 2.4)}$$

with the initial value $E_i = 3$.

For each new iteration, we have double the previous edges plus three times the number of triangles of the previous level. This follows since, for every new recursion level, each edge becomes two edges and inside each triangle three new edges are generated when the midpoints are connected. There are no edges stored, per se, but the number of edges is needed to determine how many vertices will have to be stored for subsequent iterations of the process.

The number of vertices (V) created is determined by the following recursive equation:

$$V_{i+1} = V_i + E_i \qquad \text{(equation 2.5)}$$

with initial value $V_0 = 3$.

The equation for the new number of vertices is dependent on the number of vertices of the previous iteration plus the number of edges for the previous iteration. On each of these edges, the midpoint process adds an additional vertex to the count at each recursive step. The resulting number of

vertices V_{i+1} must be multiplied by three to determine the storage required since each vertex is described by a vector of three floating point numbers. Since each floating point number requires eight bytes of storage, the total number of bytes of storage needed is 24 times the total number V_{i+1} of vertices. The number of bytes is used to determine the minimum amount of storage needed for each iteration or recursion level. This ultimately limits the number of recursive levels that can be processed. The maximum value V is determined by the minimum amount of available memory.

To solve the above recursive equations, 2.3-2.5, the initial values for the first iteration used are $T_1 = 4$, $E_1 = 9$ and $V_1 = 6$. The process of determining how many triangles, edges and vertices are formed, can be determined at each iteration. The values for the first few iterations are given in Table 2.1. The closed form solution for the number of triangles is obtained from:

$$T_0 = 1$$
$$T_1 = 4T_0 = 4$$
$$T_2 = 4T_1 = 4*4$$

It is not difficult to see that the general or closed form solution is given by

$$T_i = 4^i \qquad \text{(equation 2.6)}$$

By inspection of Table 2.1, one can see that

$$E_i = T_i + V_i - 1 \qquad \text{(equation 2.7)}$$

This relationship makes the development of the closed form solution easier. Substituting equation 2.7 into equation 2.5 one has

$$V_{i+1} = 2V_i + T_i - 1$$

Successive application of the recursion relationship yields

$$V_{i+1} = 2(2V_{i-1} + T_{i-1} - 1) + T_i - 1$$
$$= 2^2V_{i-1} + 2T_{i-1} + T_i - 1 - 2$$
$$V_{i+1} = 2^2(2V_{i-2} + T_{i-2} - 1) + 2T_{i-1} + T_i - 1 - 2$$
$$= 2^3V_{i-2} + 2^2T_{i-2} + 2T_{i-1} + T_i - 1 - 2 - 4$$
$$V_{i+1} = 2^3(2V_{i-3} + T_{i-3} - 1) + 2^2T_{i-2} + 2T_{i-1} + T_i - 1 - 2 - 4$$
$$= 2^4V_{i-3} + 2^3T_{i-3} + 2^2T_{i-2} + 2T_{i-1} + T_i - 1 - 2 - 4 - 8$$

Evaluating this for i = 3 one obtains

$$V_4 = 2^4V_0 + 2^3T_0 + 2^2T_1 + 2T_2 + T_3 - 1 - 2 - 4 - 8$$

The progression -1 -2 -4 -8 is a geometric progression and can be written as $1 - 2^4$. Leaping to a general formalation for V_i we have

$$V_i = 2^iV_0 + \sum_{j=0}^{i-1} 2^jT_{(i-1)-j} + 1 - 2^i$$

By equation 2.6 we rewrite this as

$$V_i = 2^iV_0 + \sum_{j=0}^{i-1} 2^j(4)^{(i-1)-j} + 1 - 2^i$$

$$V_i = 2^iV_0 + \sum_{j=0}^{i-1} 2^j2^{2((i-1)-j)} + 1 - 2^i$$

$$V_i = 2^iV_0 + 2^{2i-2} \sum_{j=0}^{i-1} 2^{-j} + 1 - 2^i$$

$$V_i = 2^iV_0 + 2^{2i-2} (2 - (1/2^{i-1})) + 1 - 2^i$$

$$V_i = 2^iV_0 + 2^{2i-1} - 2^{i-1} + 1 - 2^i$$

$$V_i = 2^iV_0 + 2^{2i-1} - 3(2^{i-1}) + 1 \qquad \text{(equation 2.8)}$$

Substitution into equation 2.7 yields

$$E_i = 4^i + 2^iV_0 + 2^{2i-1} - 3(2^{i-1}) \qquad \text{(equation 2.9)}$$

Equations 2.6, 2.8-2.9 gives the capability of determining how many triangles, vertices and edges, respectively, are generated at each level without needing the number at the previous level. The greatest number of levels of recursion that can be performed in the creation of fractal

mountains in this study is seven. With seven iterations the number of triangles is T_7 = 16,384, the number of edges is E_7 = 24,768 and the number of vertices is V_7 = 8385. Since each vertex is determined by a vector of three components, x, y and z, and each component requires 8 bytes of storage. One has at the seventh level a memory requirement for vertices of 24 X 8,385 or 201,140 bytes. Thus, slightly over .2 megabytes of storage is needed for storing the vertices of the base triangles that have been generated to the seventh level.

Table 2.1 NUMBER OF TRIANGLES, EDGES AND VERTICES FOR RECURSIVE EQUATIONS TO I = 7

Iteration i	Triangles T_i	Edges E_i	Vertices V_i
0	1	3	3
1	4	9	6
2	16	30	15
3	64	108	45
4	256	408	153
5	1024	1584	561
6	4096	6240	2145
7	16384	24768	8385

The stopping rule for the recursive process can be terminated optimally by stopping at the iteration that results in the best use of the available memory. The rule is then applied to the equations 2.3-2.5.

A second determining factor that could also affect the stopping criterion is a limitation on the size of the smallest addressable element on the output screen. The midpoint

displacement technique continually creates smaller and smaller triangles. The size of the triangles converge down to the smallest addressable quantity on the graphics screen, the pixel. The number of pixels on the screen will thus limit the total number of possible triangles appearing on a given computer display. If fewer iterations are performed, but the same size of the final triangle is desired, then the original base triangle must be set initially at a smaller size. The size of the triangle is measured by the number of pixels appearing on its edge. The larger the number of pixels available on a screen, the more triangles that can be created. Thus for a screen resolution of 1024 x 1024 pixels, it is possible to perform more levels of recursion then for a smaller screen resolution of say 640 x 320 pixels.

Thus, for the ultimate stopping criterion, a combination of the above factors, available memory and screen resolution is needed. The greater the number of iterations performed by the midpoint displacement technique, the more natural the appearance of the fractal mountain. Therefore, one is motivated to use the computer in an optimal way, i.e., performing as many iterations as possible.

3. Three Dimensional Geometry Based on Y Displacement

To make a realistic fractal mountain, the mountain created must be viewed in three dimensions. The image in three dimensions is created by the random displacements of

the midpoint occurring along the normal to the x-z plane, i.e. in the y direction. Changing the values in the y direction creates the appearance of upward growth as would be desired for a fractal mountain. Since the midpoint displacement is done only normal to the x-z plane, fewer calculations are needed than would be necessary if x,z displacements were also included. A further reason for displacing only along the normal to the x-z plane is the appearance of the gapping problem which is discussed below in Chapters III and IV. The gapping problem refers to breaks in the polygonal covering that represents the topography of the fractal mountain. By changing the displacement only in the y direction, the resulting gapping problem, although still difficult and important, is minimized and becomes manageable.

The displacement in the y direction helps create the natural looking fractal mountains surface. The appearance of the resulting surface is further determined by how rough or smooth a texture the surface has. In this study, a variable called 'texture' determines the smoothness of the mountain. This texture variable permits different mountain scenes, from rolling hills to rocky mountains as seen in nature, to be created on the computer screen.

4. How the Random Variables are Created for Displacing the Midpoints

We wish the amount of displacement performed at each iteration of the midpoint algorithm to be determined at random. In this section, we describe the mathematical algorithm used to obtain the randomness in the midpoint displacement technique.

The displacement of the midpoints in the y direction is based on random variables that are created with the help of a process of choosing random numbers drawn from a normal uniform distribution. The random number generator used produces a series of uniform random numbers that are then normalized to lie between zero and one. The normalized random numbers are drawn from a standard uniform distribution with mean 0 and standard deviation 1. An initial seed value is then chosen for the random number generator. The initial random number generator seed chosen for this study is an arbitrary value of 475836.

The standard uniform random variables are converted to a Gaussian distribution and are placed into an array called RAND in the order that they are generated. The final array RAND is a table of 500 Gaussian distributed random numbers with mean zero and a standard deviation of one. The random number chosen for the displacement from array RAND is determined by a variable "loc". Loc determines the value for the number chosen from array RAND and ranges from 1 to 500.

The value of loc is important in the algorithm due to the two fundamental problems that occur in achieving natural looking fractal mountains. These two fundamental problems of granularity and gapping are described and defined in the next chapter. Following that description, in Chapter IV a description of the impact of the granularity and the gapping problems on the algorithms used in the implementation of the random midpoint displacement technique is made.

III. PROBLEMS AFFECTING REALISM

Two problems affect the realism we see in the creation of fractal mountains. The problems are granularity and gapping.

A. GRANULARITY

A major concern in creating fractal mountains is the realism of the resulting picture of the mountain. The appearance of a natural look in the picture of a mountain is determined in part by the texture given by the algorithm that creates the mountain. The extremely regular pattern consisting of rows of valleys and ridges shown in the fractal mountain in Figure 3.1 is not representative of actual mountains. This regular pattern is called granularity. The more patterned or granular a fractal mountain is, the less realistic the mountain appears. To eliminate the granularity as shown in the fractal mountain of Figure 3.1, more randomness must be used in the midpoint displacement technique. The more random the midpoint displacements produced by the algorithm, the less granular will be the appearance of the fractal mountains. For example, the technique used to produce the mountain in Figure 3.1 used only 8 different random variables. The methods used to minimize the granularity by the introduction of more randomness are discussed in Chapter IV. Additionally, the

granularity is affected by the orientation of the triangles An important factor affecting the granularity is how the subroutine calls are made for each triangle being operated on by the midpoint displacement technique.

B. GAPPING

Another problem that affects the natural look of the fractal mountain is the "gapping problem". The gapping problem occurs when one applies the midpoint displacement technique to the common side of two adjacent triangles. Referring to Figure 3.2a, the two adjacent triangles are A and

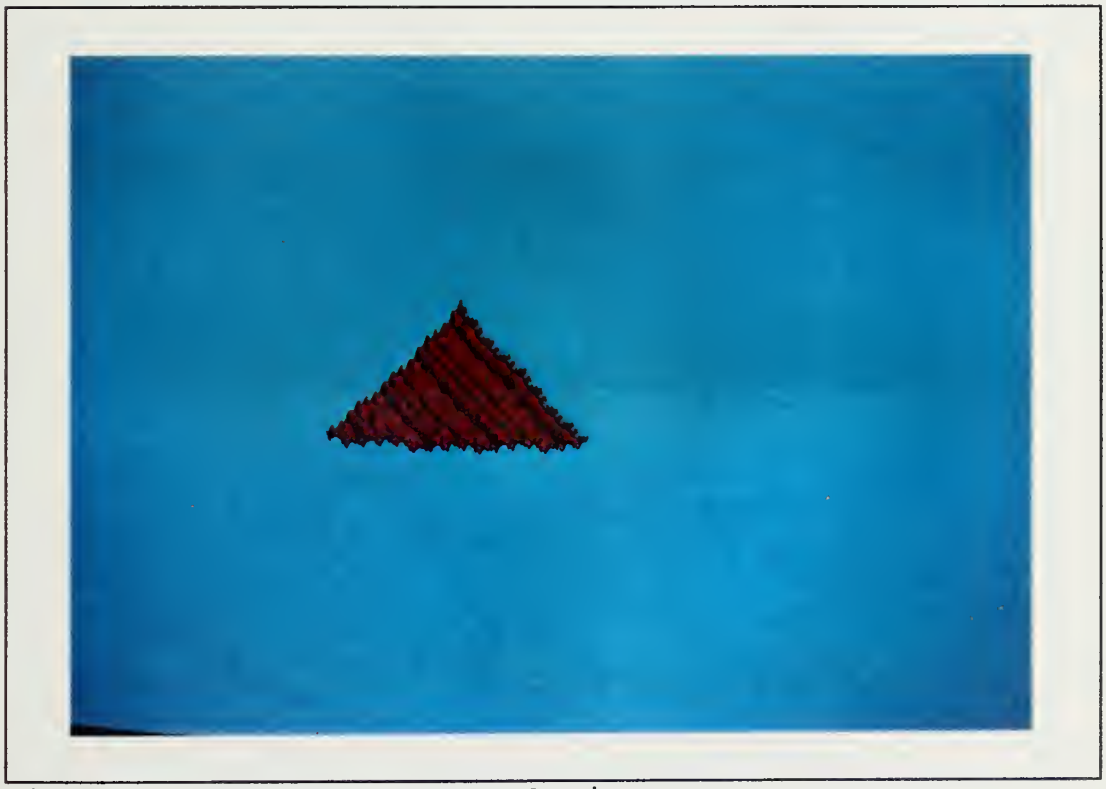

Figure 3.1 Example of Granularity

B. They share the common side P_3P_4 with the midpoint indicated by M. The midpoint displacement technique displaces M first in triangle A and then in triangle B. In Figure 3.2b the point M has been displaced in triangle A to point M_A and in triangle B to point M_B. The resulting polygon $P_3M_AP_4M_B$ defines a gap or discontinuity in the figure. In order to eliminate the gap, the vector displacement of M in triangle A must be the same as the vector displacement of M in triangle B. Chapter IV discusses the technique used to insure displacements without a gap.

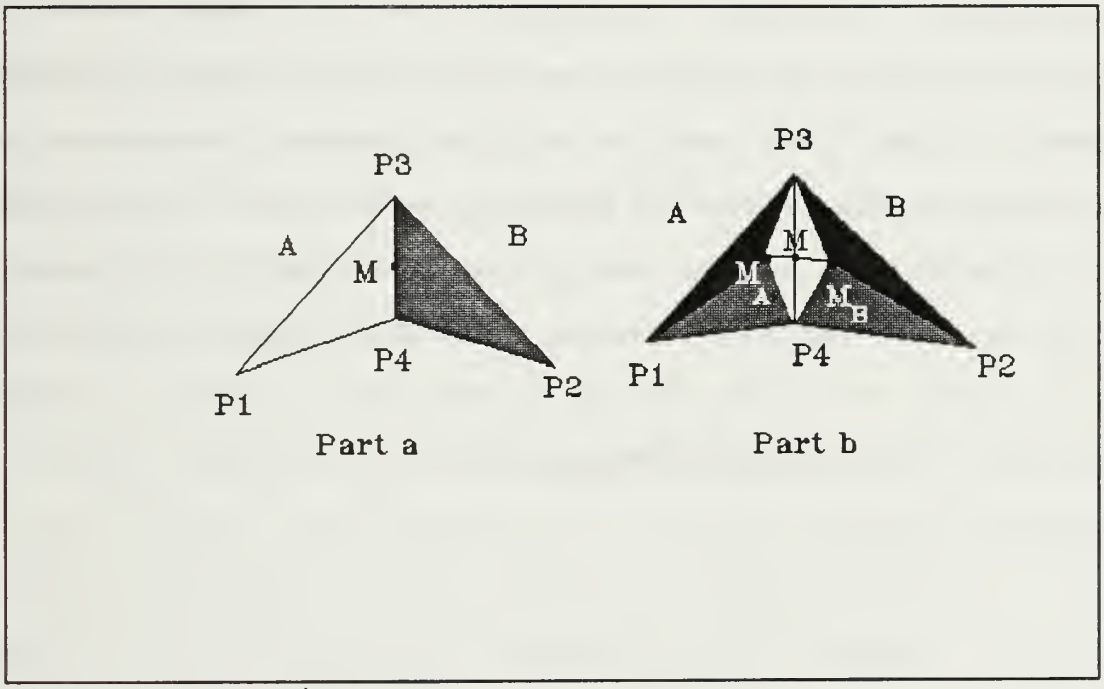

Figure 3.2 Gapping Problem

C. INTERRELATIONSHIP BETWEEN GRANULARITY AND GAPPING

In order to eliminate the problem of granularity in the creation of fractal mountains, randomness is needed in the midpoint displacements, which also results in a realistic mountain. On the other hand, the problem of gapping from two different midpoint displacements of the common side of adjacent triangles demands that the two displacements of a midpoint on a common side be equal. This means a certain regularity has to be imposed on the midpoint displacement technique. A conflict results between the solution of the granularity problem and the solution of the gapping problem. Thus, a trade off has to be made between randomness and regularity . The effect of granularity and that of gapping and the combination of the two problems is further discussed in Chapter IV. The simultaneous solution, finally adopted for this study, for both the granularity problem and the gapping problem is presented in Chapter V.

IV. PUTATIVE ALGORITHMS FOR THE GRANULARITY AND GAPPING PROBLEMS

In Chapter III, the problems of granularity and gapping affecting the naturalistic look of a fractal mountain were defined and briefly discussed. This chapter discusses the algorithms that have been considered to solve the granularity problem and the gapping problem. The various approaches discussed here did not lead to a simultaneous solution to both of the problems. The discussion of these algorithms is included because it seems useful to do so. The discussion shows the complications that can occur in an effort to create natural looking mountains. The solution that simultaneously handles the granularity problem and the gapping problem, incorporating some of the approaches used in this Chapter, is presented in Chapter V.

A. RANDOM VARIABLE EFFECTS ON GRANULARITY

The granular appearance of the fractal mountain depends on the randomness of the midpoint displacements. It is possible the resulting granularity is caused by some undiscovered regularity in the random variables generated by the random number generators. To determine if the granularity is due to the quality of the random variables, the Gaussian

distribution random number generator, described in chapter II, was replaced by a linear congruential random number generator.

The linear congruential method creates a sequence, a(i), of random numbers by using the following equation:

$$a(i) = ((a(i-1) * b) + 1) \bmod m \qquad \text{(equation 4.1)}$$

An arbitrary initial value a(0) is needed to start equation 4.1. Constants b and m for the multiplier and the modulus, respectively, are also used in the process. Each different initial value yields a different random sequence. The next random number in the sequence is obtained from a(0) by multiplying a(0) by the constant multiplier b and adding 1. The operation [(a(0)*b)+1] is performed modulo the second constant m. The resulting value is taken as the second random number a(1). The process continues, producing a(2),a(3),...,a(499). The constants m and b are chosen subject to some constraints. Sorensen [ref. 20] states that the best choice for m is a power of 10 or a power of 2 depending upon the index of the number system used. Further he states that b should be a number one digit less than m. An additional constraint for b is that it be of the form x21 where x is an even number. For example with m=1024, and with x arbitrarily chosen as 8, b becomes 821. (notice m has four digits and b has three digits). The resulting random number sequence is composed of integers between 0 and m-1. For this

random number sequence to be used in the midpoint displacement technique, the numbers need to be normalized to lie between [0,1].

Using the random numbers produced by the linear congruential method instead of those produced by the normal uniform distribution method, discussed in Chapter II, does not materially change the granular texture observed in Figure 3.1. It is therefore concluded that the granularity problem is not due to the quality of the numbers produced by the random number generator. The normal uniform distribution generator is therefore retained in the actual implementation. It is therefore further concluded that the granularity problem is the result of some different aspect of the implementation of the midpoint displacement technique.

To understand the effect of the random variables on granularity, the effect of the selection of the variable **loc** was investigated. The variable **loc** is used to choose the random number chosen from the array RAND. To begin the process, **loc** can be arbitrarily chosen equal to 1. The first random variable used is thus the random variable in the first position of array RAND. When the base triangle P1 P2 P3 is operated on by the random midpoint displacement technique, the midpoints Pmid1, Pmid2 and Pmid3 are created (see Fig 4.1). The first midpoint displaced is Pmid1, so the displacement of

Pmid1 in the y direction uses only the first number from RAND (i.e., loc = 1). A predetermined value of the variable texture, as described in Chapter II, is also incorporated in the equation:

$$\hat{Pmid1}(y) = (texture * RAND(loc)) + Pmid1(y) \quad \text{(equation 4.2)}$$

For each of the succeeding sides in the triangle, the pointer loc is successively incremented by 1. One obtains the equations:

$$\hat{Pmid2}(y) = (texture * RAND(loc+1)) + Pmid2(y) \quad \text{(equation 4.3)}$$

$$\hat{Pmid3}(y) = (texture * RAND(loc+2)) + Pmid3(y) \quad \text{(equation 4.4)}$$

Note: the hatted variables are updated versions of the previous variables. The hats shall be suppressed in the sequel.

The random displacements of the midpoints is a recursive process. We also update the random numbers drawn from RAND. At the next level of recursion, we increment the value of the pointer in the table RAND (i.e., loc = loc + 1). For the next set of midpoint displacements, the pointer loc is set equal to the second, third and fourth positions of RAND. This process is repeated through each level of iteration (See Fig 4.1). The positions from RAND used for each midpoint are denoted by arabic numbers. For six iterations, there are up to eight different random variables chosen from array RAND. In this case, the random variables are those in locations 1

to 8 of the array RAND. The process shown which makes smaller triangles is also performed for triangles II, III and IV. Other smaller triangles are created until the triangles are as small as a predetermined size. This yields a structure such as seen in the top triangle of Figure 4.1. The regular pattern of the numbers as seen in Figure 4.1, i.e., the selection of locations of RAND indicates that the midpoint displacements are not truly random. This method of selecting loc leads directly to the granularity of Figure 3.1. Thus, somehow the regularity in the selection of the random variables needs to be avoided.

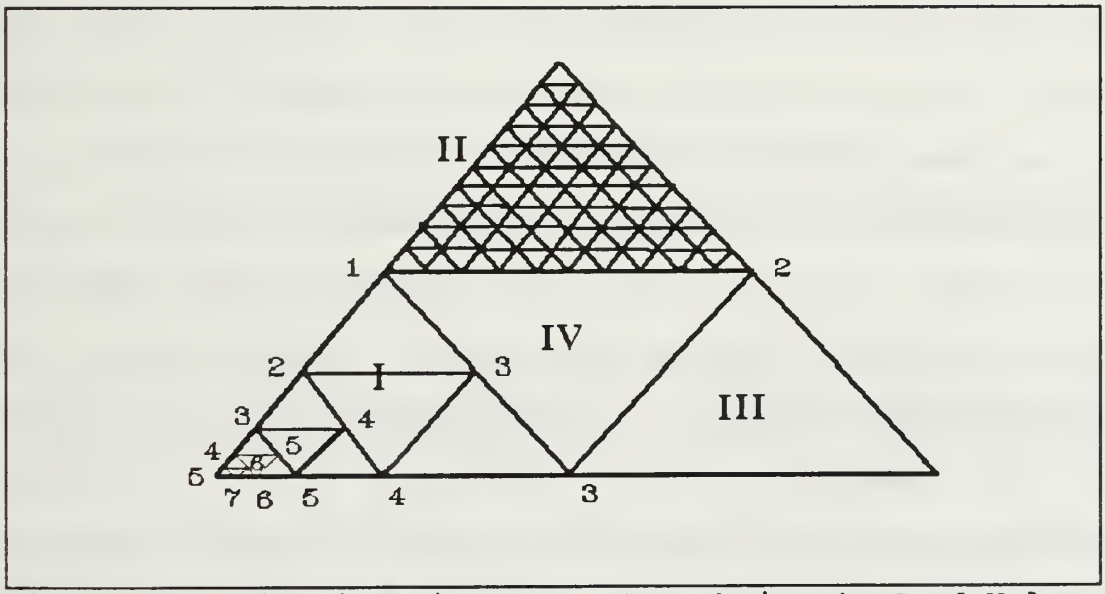

Figure 4.1 Midpoint Displacement Technique's Seed Values

B. MODIFICATION OF THE MIDPOINT DISPLACEMENT TECHNIQUE TO ELIMINATE GRANULARITY

As the random number generator has been shown to not have a deleterious effect on the granularity problem, other solutions for eliminating the granularity need to be considered. Both the effect of weighted displacements of the midpoint and the effect of small based triangles were considered in order to eliminate granularity.

1. Weighted Displacements of the Midpoints

The first modification to the midpoint displacement technique considered is to create weighted displacement of the midpoints. The term weighted displacement is used to describe different maximum displacements of the midpoints as a function of the size of the initiator (the line segment) at each step. When the size of the initiator is smaller than a predetermined value, the maximum displacement of the midpoint is reduced in the weighted midpoint displacement technique. The original implementation of the midpoint displacement technique used in this study, was taken from work of Michael Gaddis [ref. 21] and is described above as the original implementation. This implementation of the midpoint displacement technique resulted in the granularity shown in Figure 3.1. A detailed description of this algorithm is needed in order to understand fully the modification of weighted midpoint displacements

developed for solving the granularity problem.

Original Algorithm (Equations 4.5-4.10)

To determine the displaced midpoint in the first implementation of the midpoint displacement technique, the midpoint of the y values between two endpoints of the initiator is first found.

Initialization Equations (Equations 4.5-4.7)

The equations to determine the midpoints for the three edges of the triangle first find the midpoint of the x, y and z values from the endpoints. The equations are:

$$Pmid1(x) = (P1(x) + P2(x))/2.0 \quad \text{(equation 4.5)}$$
$$Pmid2(x) = (P2(x) + P3(x))/2.0 \quad \text{(equation 4.6)}$$
$$Pmid3(x) = (P3(x) + P1(x))/2.0 \quad \text{(equation 4.7)}$$

Similar equations are usedto find the three midpoints in terms of their y and z coordinates. Once the midpoint on each edge is found, the y coordinates of the midpoints are randomly displaced.

Displaced Equations (Equations 4.8-4.10)

The random displacement for the three midpoints in the y direction are given as:

$$Pmid1(y) = (texture * RAND(loc)) + Pmid1(y) \quad \text{(equation 4.8)}$$
$$Pmid2(y) = (texture * RAND(loc+1)) + Pmid2(y) \quad \text{(equation 4.9)}$$
$$Pmid3(y) = (texture * RAND(loc+2)) + Pmid3(y) \quad \text{(equation 4.10)}$$

Texture (described in Chapter II.B.3) is the roughness factor and takes on a value in a range between 1 and 20 to

give the roughness or smoothness of the surface. RAND(loc) is explained in Chapter II.B.4 and is the random number used to determine the amount of displacement of the midpoint value. Note three different locations are chosen for the three equations. The displacement of the midpoint of each edge, Pmid1(y), Pmid2(y) and Pmid3(y) are then determined by the equations 4.8, 4.9 and 4.10, respectively.

Weighted Displacement Algorithm (Equations 4.11-4.16)

In the modified algorithm, where weighted displacement of the midpoints is attempted, the initial equations 4.5, 4.6 and 4.7 are retained, but the displacement equations 4.8, 4.9 and 4.10 are altered. The difference in the equations used is determined by the distance between the two endpoints of the two points in the x and z coordinates, measured in pixels. For each of the endpoints, the distance is determined. The equations are:

Selection Equations (Equations 4.11-4.13)

$$Pab1 = \text{sqrt}[(P1[x]-P2[x])^2 + (P1[z]-P2[z])^2] \quad \text{(equation 4.11)}$$

$$Pab2 = \text{sqrt}[(P2[x]-P3[x])^2 + (P2[z]-P3[z])^2] \quad \text{(equation 4.12)}$$

$$Pab3 = \text{sqrt}[(P3[x]-P1[x])^2 + (P3[z]-P1[z])^2] \quad \text{(equation 4.13)}$$

Pab1, Pab2 and Pab3 are used in the criteria to determine whether the displaced equations are to be modified. If any of the values Pab1, Pab2 or Pab3 are greater than a selected value ϵ, then the respective equations used are the

original displaced equations 4.8, 4.9 and 4.10 for the given value. The use of equations 4.8-4.10 allows a large displacement in the y direction, in general. As the distance between the points measured in the x and z coordinates gets smaller, the possible displacement needs to be smaller than when the size of the initiator is large. This change in displacement prevents the possibility of a valley and a peak in a mountain scene being right next to each other. This can occur naturally, for example, in a rift valley setting. For now we seek more ordinary mountain settings.

If, on the other hand, any of Pab1, Pab2 or Pab3 are less than or equal to ϵ, then the respective equations used for displacement are a function of the distance of the x and z coordinates:

Modified Displaced Equations (Equations 4.14-4.16)

$$Pmid1(y)=(RAND(loc)*texture*(Pab1/\epsilon)) + Pmid1[y] \quad \text{(equation 4.14)}$$

$$Pmid2(y)=(RAND(loc+1)*texture*(Pab2/\epsilon)) + Pmid2[y] \quad \text{(equation 4.15)}$$

$$Pmid3(y)=(RAND(loc+2)*texture*(Pab3/\epsilon)) + Pmid3[y] \quad \text{(equation 4.16)}$$

Here, ϵ is set to 50.0 and loc is the location of the random number chosen from the array RAND.

ϵ is set to 50.0 to allow the number of edges of each triangle to occur at different recursion levels if the

triangle is an acute, an equilateral or an obtuse triangle. The ending criteria (see Chapter VI.D) is set at either 16 pixels or 9 pixels for the length of an edge of a triangle. Because edge lengths are successively halved at each iteration, for the edge of a triangle to be less than 16 pixels, the resulting edge length would be strictly between 8 and 16. Suppose the edge length when the stopping criterion is met occurs at the fifth recursion level. Then at the fourth recursion level there would have to be an edge of length between 16 and 32. The third recursion level would have edge lengths between 32 and 64. The value of 50.0 lies approximately 2/3 of the way from the value of 32 to the value of 64. Therefore, if the base triangle is chosen to have edges of length between 250-500 pixels, as is common in this study, then the following will occur for the three types of triangles. If the triangle is an acute triangle when the third recursion level is completed, one of the three edges would be under 50 pixels in length and therefore the weighted displacement equation would be used in the next iteration on that edge. Then on the fourth recursion level the other two edges would also use the weighted displacement equations. If the triangle is an equilateral triangle, the three edges of the triangle would all use the weighted displacement equations at the same recursion level. If all sides are less than 350

pixels in length, the recursion level at which the weighted displacement equations are used would be the third level. If the triangle is an obtuse triangle, then at the third recursion level two of the three edges would use the weighted displacement equations. The third edge would use the weighted displacement equation at the fourth recursion level. The results for the acute, equilateral and obtuse triangles makes ϵ equal to 50.0 an attractive choice.

If, for example, only Pab1 is less than ϵ, then equation 4.8 is replaced by equation 4.14 and the equations 4.9 and 4.10 are used in the other two respective cases. Corresponding changes for Pab2 and Pab3 are made if these numbers are less than ϵ.

Using the weighted displacement algorithm for the midpoints gives only a slight improvement in the realism of the fractal mountain. Although there was less granularity in the picture corresponding to Figure 3.1, the fractal mountain still did not look very realistic. The weighted displacement technique of the midpoints is thus not the final solution for improving the realism of the fractal mountains, so further modifications were sought. However, since there was some improvement with the weighted displacement technique, this technique is incorporated in the implementation used to achieve the naturalistic effect in fractal mountains.

2. Smaller Base Triangles

The next attempt to remove the granularity is to use smaller base triangles. The original implementation began with a base triangle with a couple hundred pixels in length for each edge the size seen in Figure 3.1. The original base triangle is then divided into four arbitrary sized triangles of about 40-50 pixels for each edge as seen in Figure 4.2. These four new triangles are then used as base triangles in the algorithm implementing the midpoint displacement technique. The resulting fractal mountain using the smaller base triangles is seen in Figure 4.3. The four triangles each have a different granular appearance. On the right, where triangles III and IV join, small gaps can be seen. There are also gaps between each of the pairs of triangles, but on only the one common side of adjacent triangles are the gaps large enough to be seen without magnification. Each triangle still exhibits a distinctive granular appearance. Because there are four different granularities, the fractal mountain looks more realistic compared to the original fractal mountain as shown in Figure 3.1. This solution of the granularity problem with small base triangles, however, introduces a gapping problem.

C. WHY THE ORIGINAL IMPLEMENTATION DID NOT RESULT IN GAPS

In sections 4A and 4B, the use of different random variables and smaller base triangle sizes were considered to deal with the granularity problem. In section 4B, smaller triangles helped solve the granularity, but introduced gaps into the solution of the midpoint displacement technique. In this section, the gapping problem is further considered. The fractal mountain of the original implementation seen in Figure 3.1 contains no gaps. Although the gapping problem did not exist within the original implementation of the midpoint displacement technique, we have seen that efforts to create better granularity can introduce a gapping problem. A description of the algorithm in the original implementation, may help in an understanding of the modifications needed for the algorithm to solve both the granularity problem and the gapping problem simultaneously.

As discussed in Chapter II, the creation of fractal mountains is accomplished recursively. The recursion process is fundamental for the creation of fractal mountains. But just as important, for realistic fractal mountains is the implementation of the recursion process. In the original implementation, the subroutine used to implement the midpoint displacement technique is called mountain_generate. By the

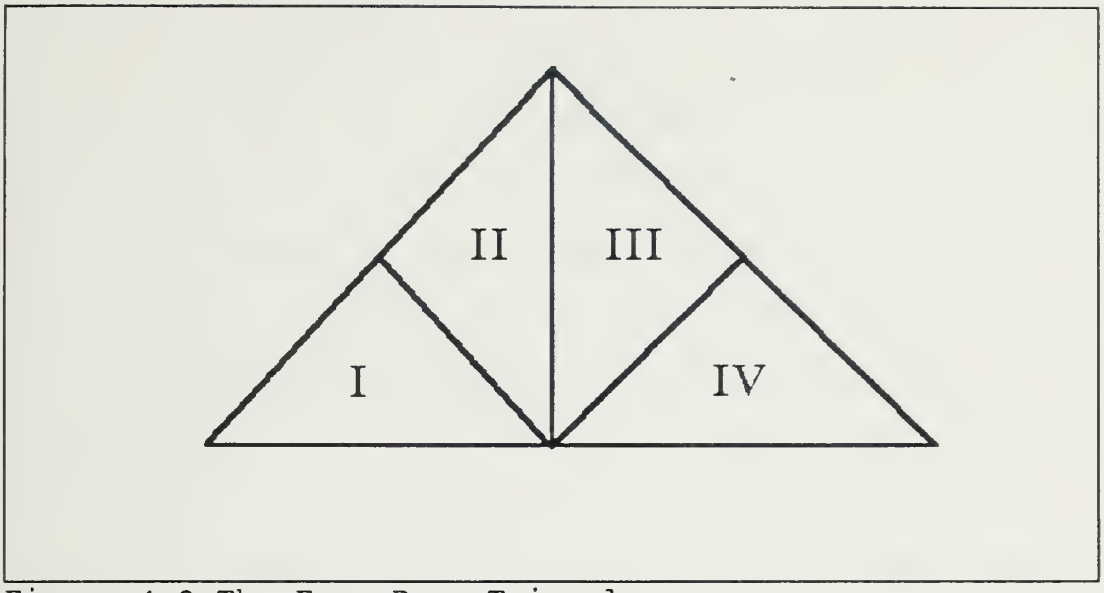

Figure 4.2 The Four Base Triangles


Figure 4.3 Fractal Mountain using Small Base Triangles

process of recursion, the mountain_generate algorithm is called from within mountain_generate. As seen in Figure 4.4, Triangle P1 P2 P3, when processed by the subroutine mountain_generate, is divided into four new smaller triangles within the original triangle. Each of the new triangles forms the basis for a new level of recursion. The process continues until some smallest acceptable triangle size is reached by the stopping criterion. This results in the triangle structure such as seen in the top triangle of Figure 4.1. In Figure 4.4, the new midpoints Pmid1, Pmid2 and Pmid3 when connected create the four new triangles I, II, III and IV. Notice triangle I is in the upper part of the original base triangle P1 P2 P3. The orientation of each triangle I, II, III and IV is important in achieving the desired end of eliminating the gapping problem. The triangles have to be rotated in an appropriate way for each call to mountain_generate.

To implement the recursive call for the midpoint displacement technique on each of the triangles, I, II, III and IV, a call to mountain_generate is used. The mountain_generate call uses the three endpoints of the respective triangle as input variables and a fourth input variable, for indicating the location in RAND for the value for the random number used. The order of the three vertices of the triangles in the call is crucial. Referring to Figure

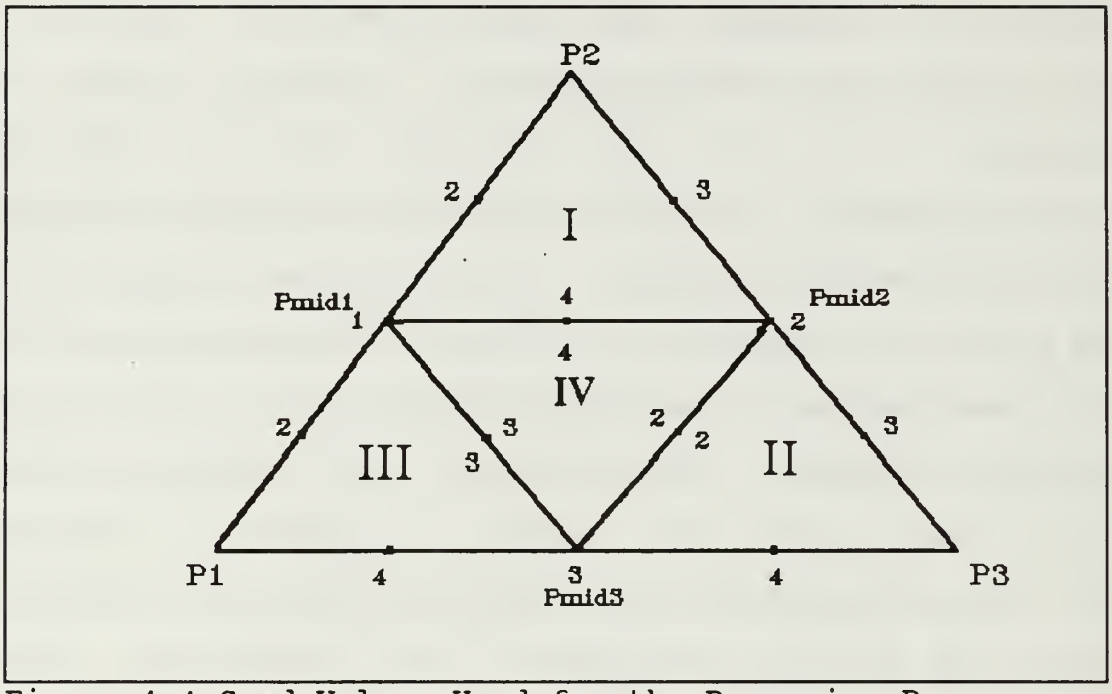

Figure 4.4 Seed Values Used for the Recursive Process

4.4 for each triangle, the following recursive calls to mountain_generate are used.

Recursive Calls

```
mountain_generate(P1,P2,P3,loc);      for Base Triangle
loc = loc + 1;
mountain_generate(Pmid1,P2,Pmid2,loc);  for Triangle I
mountain_generate(Pmid3,Pmid2,P3,loc);  for Triangle II
mountain_generate(P1,Pmid1,Pmid3,loc); for Triangle III
mountain_generate(Pmid2,Pmid3,Pmid1,loc);
                                        for Triangle IV
```

As can be seen from Figure 4.4, the above recursive call for triangle I starts with the left endpoint Pmid1 as the first variable, the uppermost endpoint P2 for the second variable and the right most endpoint Pmid2 for the third

variable. The first, second and third endpoints are determined as the corresponding first, second and third variables sent to the routine mountain_generate. The order of the input variables to mountain_generate is important, as this order determines which value of loc is used on each edge of the triangle for displacing the midpoints. The second, third and fourth recursive calls rotate the triangles II, III and IV so that appropriate loc values are used on the common edges of adjacent triangles.

Rotation of the triangle is accomplished by the order in which the endpoints are presented in the mountain_generate call. As indicated previously, triangle P1 P2 P3 forms the first level of recursion, while triangles I, II, III and IV are in the second level of recursion. For each level of the recursion, the same three consecutive values are used for loc. Thus, values of loc equal to 1, 2 and 3 are used in the first level of recursion. Loc is then incremented by 1. For every triangle created in the second level of recursion the values of loc are taken to be 2,3 and 4. These values are denoted by the numbers on the respective edges in Figure 4.4. Specifically, in the call for triangle I in Figure 4.4, loc will be 2 for the midpoint between the first endpoint Pmid1 and the second endpoint P2. Similarly, loc is 3 for the midpoint between the second endpoint P2 and the third endpoint

Pmid2. The last edge of the triangle has loc equal to 4 used for displacing the midpoint between the third endpoint Pmid2 and the first endpoint Pmid1. As can be seen in the recursive calls, the same values of loc i.e., 2, 3 and 4, are used three more times for each of the triangles II, III and IV in Figure 4.4. The value of loc used for triangles II, III and IV is determined in a similar manner as that described for triangle I using the last three recursive calls shown above. Triangles I, II and III are oriented so that the common sides of the adjacent triangles have the same loc and therefore, the same displacement. This matching of common sides of adjacent triangles yields a solution to the gapping problem because the same random number is used in the displacement process for both calls of the algorithm on a common side.

Notice however that only four different loc values are used. And each value is repeated four times at the given level of the recursion in the algorithm. Repeated use of the same value of the loc implies that the displacement of the midpoints is not completely random as discussed previously. In addition, there is regularity or granularity introduced in the fractal mountain by the orientation pattern of the triangle. Although the original implementation has no gaps, it does tend to perpetuate the granularity problem.

In order to solve simultaneously the granularity problem and the gapping problem, careful attention to detail is required. Thus not only is the basic technique used to create fractal mountains important, but also the actual implementation of the midpoint displacement technique is important. The implementation is important in order to insure true randomness with no gaps in the creation of the fractal mountain. In the next chapter, the algorithms finally developed to solve both problems are given. That is, the gaps are controlled and an amount of granularity that can be called acceptable is achieved.

V. IMPLEMENTATION USED TO ACHIEVE A NATURALISTIC EFFECT IN THE CREATION OF FRACTAL MOUNTAINS

This chapter explains the implementation that ensures a naturalistic look to fractal mountains. The first section explains how the random variables are used to achieve a naturalistic look. In section B, the hidden surface removal and the lighting calculations are discussed. In the last section, the algorithms dealing with the intersections of the outline for the mountain range are discussed.

A. HOW RANDOM VARIABLES ARE USED TO CREATE A NATURALISTIC LOOKING FRACTAL MOUNTAIN

Random variables are used to create in the fractal mountain the realism seen in nature. As was seen in Chapter IV, the modifications to the algorithms using the midpoint displacement technique showed that the technique was important in producing realistic mountains. Also, unless great care is taken with the implementation of the technique, the simultaneous solution of the granularity and gapping problems cannot be achieved. As was discussed in Chapters III and IV, there is a trade off between the degree of randomness employed and the degree of regularity used. On the one hand, to solve the granularity problem, a great deal of randomness is needed

in the midpoint displacements. On the other hand, the gapping problem requires a certain amount of regularity in displacing the midpoints.

The random variables used in the program are formed into a uniform normal distribution. To solve the gapping problem, the midpoint displacement on the occasions when the common side of two adjacent triangles are considered, must be the same. The two endpoints of the common side of adjacent triangles are the same and the values of x and z coordinates are not displaced by the algorithm. Therefore, the value of loc, used to determine the random variable to displace the midpoint in the y direction, is chosen to be a function only of the x and z coordinates of the two endpoints. In this way, the displacements of the same point in the two triangles is never different. Thus, no gaps can develop.

The following equations (5.1-5.3) are used to determine the value of the location in RAND for each of the midpoints of the edges of the triangle. To determine the value of loc, the two x values and the two z values are added. Addition is used for the two endpoints instead of a subtraction so that the value is not dependent on the order in which the vertices are chosen. Since the value should range between 0 and 499 the addition of the x and the z coordinates are modulo 500.

For displacing the midpoint Pmid1, as seen in Figure 4.4 the equation is:

```
PT = (P1[x]+P2[x])+(P1[z]+P2[z])
loc = PT mod 500                                (equation 5.1)
```

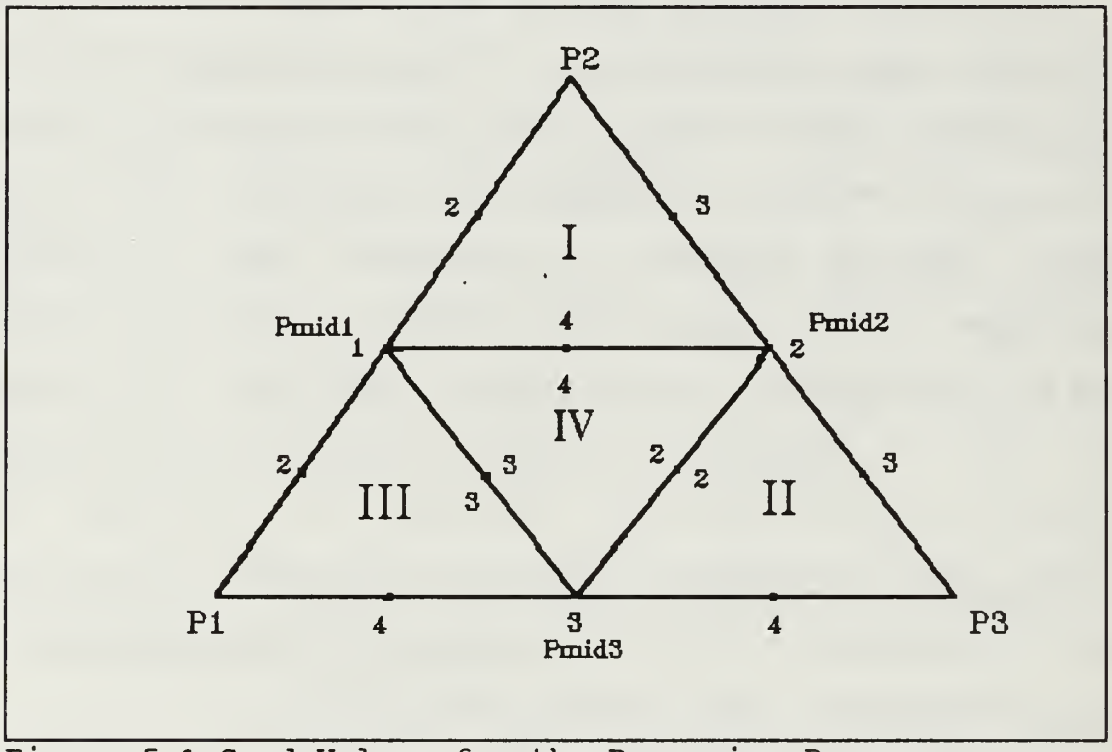

Figure 5.1 Seed Values for the Recursion Process

For displacing the midpoint Pmid2 the equation is:

```
PT = (P2[x]+P3[x])+(P2[z]+P3[z])
loc = PT mod 500                                (equation 5.2)
```

and the equation for midpoint Pmid3 is

```
PT = (P1[x]+P3[x])+(P1[z]+P3[z])
loc = PT mod 500                                (equation 5.3)
```

Since there are 500 values in array RAND, the value of loc, must be between 0 and 499. Using the modular computation routine gives a good distribution of numbers from 0 to 499.

In Chapter IV, it was shown that in the original implementation the triangles in the iterative procedure were oriented a certain way to prevent the appearance of gaps. Equations 5.1-5.3 solve the gapping problem without having to turn the triangles for each level in any particular orientation as was needed in the original implementation. The mountain_generate calls for the recursion process are the following:

```
mountain_generate(Pmid1,P2,Pmid2,ilev)
mountain_generate(P3,Pmid3,Pmid2,ilev)
mountain_generate(Pmid1,Pmid3,P1,ilev)
mountain_generate(Pmid1,Pmid3,Pmid2,ilev)
```

Notice in the above calls to mountain_generate that the triangles are presented to mountain_generate in one order. Since the order does not matter, the endpoints above are only one order that can be used. All combinations of the three endpoints are possible. The variable **ilev** is used to determine the recursion level being executed. Ilev is used so that if there is more than one base triangle, all the base triangles will have the same number of recursion levels and the final iteration will result in triangles of the same approximate size.

Although there is regularity in displacing the midpoint of the common side of two adjacent triangles, there is also enough randomness to prevent the patterns characteristic of the granularity problem. That is, the variable loc is computed in such a way to allow any of the values of RAND to

be accessed. The original implementation used only 8 random variables in processing through five levels of recursion. The present implementation uses all of the 500 random variables in RAND at least in theory. In the case where the recursion level results in more than 500 midpoint displacements, there will necessarily be repeated random variables. This repetition will not occur in any regular pattern and therefore will not result in regularity or granularity appearing in the fractal mountain.

Another idea providing randomness is that of weighted displacement (Chapter IV.B). The idea of weighted displacement has been implemented to make the mountains as naturalistic looking as possible.

With the simultaneous solution of the granularity and gapping problems to give the desired realism to the fractal mountains, the subject of hidden surface removal and of the mathematical calculations for lighting can be explained. These two subjects are important components in producing a natural look in the display of fractal mountains.

B. HIDDEN SURFACE ELIMINATION AND LIGHTING CALCULATION

Hidden surface elimination and sophisticated lighting routines can be used to give fractal mountains a natural look. Z-buffering is used to achieve the hidden surface elimination and the lighting is used to provide the effect of different times of day.

1. How Z-buffering Adds to Fractal Mountains

Since fractal mountains are being created in three-dimensions in our study, there is a need to develop the concept of hidden surface elimination in the algorithm for the computer being used. The z-axis is portrayed as being perpendicular to the screen or to the xy plane. The concept used is called Z-buffering. [ref 22] Z-buffering adds to the naturalistic look by achieving the hidden surface elimination in the fractal mountain.

2. Mathematical Calculations for Lighting

Lighting and shading also affect the naturalistic look of fractal mountains. We use the IRIS workstation lighting capabilities. The creation of a fractal mountain requires the operator to provide a definition of 1) the color and the surface properties of the fractal mountain, 2) the color, location and direction of the light source and 3) the position and view direction of the observer. The surface properties for the fractal mountains are characterized by three factors. These are ambient light, diffuse light and specular light. Ambient light is a light of uniform brightness caused by multiple reflections from many surfaces of the fractal mountain and is non-directional. Light from ambient and point sources, which is scattered in all directions by the surfaces, is called diffuse light. It results from the surface roughness or graininess. Point sources create highlights

called specular reflection on shiny surfaces. Since specular light is mainly from smooth surfaces very little specular light is used for general fractal mountains. We note an exception in the next chapter. The ambient light and diffuse light vary according to the type of color and surface desired for the fractal mountain texture. Each of the three light types is specified by a red, a green and a blue component. Each component is given a value between 0 and 1. A forested mountain, for example, would have a large green component and small red and blue components.

Characterization of the light source is an important factor in the lighting. The light source, which is taken to be the sun, consists of an ambient light component, a color component and the position of the light. The ambient light and color are specified by red, green and blue values between 0.0 and 1.0 as in the specification of the mountain surface light properties. The color of the sun is again specified by its red, green and blue components. The position of the light source or the sun is determined by a four component vector with values given by x, y, z and w. The w value determines if the light source is local or at infinity. If the light source is local, w is set equal to 1.0 and if w is set equal to 0.0, the light source is taken to be at infinity. When the light source is taken to be the sun, it is always placed at infinity. The x, y and z values determine the position of the light in the local case or the

direction of the light source when the light source is set at infinity. The x, y and z values for the location of the light source can be varied to achieve morning, noon or evening lighting effects.

An additional property of the lighting consists of the overall level of light falling on the entire scene. This is referred to as scene ambient light. The scene ambient light parameter controls the general level of light in the fractal mountain scene. When the scene ambient light is low such as at morning, the whole scene is darker than when the scene ambient light is high such as at noon. To define ambient light, the three values of the red, green and blue components between 0.0 and 1.0 must be given. The location of the viewer is also a variable in the light calculations. To determine if the viewer is a local viewer or a viewer seated at infinity, a logical variable must be set for the program. This logical variable is 1.0 for TRUE if the viewer is local and 0.0 for FALSE if the viewer is at infinity.

Once the above properties of the mountain surface, the light source and the scene light have been defined, the lighting calculations for the fractal mountain can be made. To make the calculations, the light perceived by the viewer depends on the angular relationships between the light source direction, the view direction and the surface normals. The light source direction is the position of the sun as specified in the program with respect to the surface of the mountains.

The view direction is the direction of the observer. The surface normal is the orientation of the surface of each triangle that is used in creating the fractal mountain. The normal is perpendicular to the surface of each triangle. This surface normal is a vector and is calculated from the vertices of the triangle by using the dot product, as shown in the following equations from the program.

```
NLS[0]=((P2[y]-P1[y])*(P3[z]-P1[z]))-((P3[y]-P1[y])*(P2[z]-
    P1[z]))
NLS[1]=((P2[z]-P1[z])*(P3[x]-P1[x]))-((P3[z]-P1[z])*(P2[x]-
    P1[x]))
NLS[2]=((P2[x]-P1[x])*(P3[y]-P1[y]))-((P3[x]-P1[x])*(P2[y]-
    P1[y]))
```

The surface normal vectors needs to be of unit length. Thus, x**2 + y**2 + z**2 equals 1. With x = NLS[0], y = NLS[1] and z = NLS[2] the normalized surface vector is given by

```
NL1[0] = NLS[0]/sqrt((NLS[0]**2) + (NLS[1]**2) + (NLS[2]**2))
NL1[1] = NLS[1]/sqrt((NLS[0]**2) + (NLS[1]**2) + (NLS[2]**2))
NL1[2] = NLS[2]/sqrt((NLS[0]**2) + (NLS[1]**2) + (NLS[2]**2))
```

When the viewer is positioned at infinity and the light source is at infinity, as is the typical case for fractal mountains, only one normal vector for each triangle needs to be given. The calculations for lighting the fractal mountain are all done automatically by the software provided with the IRIS workstation. Once the surface properties, the light source and the light model have been defined and the normal is sent to the workstation, the calculations for the color of

the triangle can be determined. When the light source position changes such as from morning to evening, different effects will appear on the fractal mountain which helps create a realistic image.

C. MOUNTAIN RANGE OUTLINE

The mountain range outline can be drawn anywhere within the window. Since one can draw anywhere within the window, there must be some routine that checks for intersections and duplicated y values for each x value. A description of this routine follows.

Each point of the outline is connected by a straight line with the previous point. Each pixel with an x coordinate must have a unique y coordinate. To accomplish one y coordinate for each x coordinate, an interpolation must occur between pairs of points. The interpolation is done by first determining the slope between the two points. Starting with point (x_1,y_1) and point (x_2,y_2), the slope is computed as:

$$sl = (y_2 - y_1) / (x_2 - x_1) \qquad \text{(equation 5.4)}$$

Once the slope has been determined, the equations used to determine the y coordinate for each x coordinate are determined by:

$$x_i = i$$

$$y_i = (sl * x_i) + y_1 \qquad \text{(equation 5.5)}$$

Here, i takes on values in the range of 0 to 1049.

Once the array y_i is filled, the values computed and placed in the array are compared to all the values given when creating the outline. If there are duplicate y coordinates for a given x coordinate, then the larger value of y is used. The result of this operation is to remove any back tracking or any intersections. When one of these pairs of duplicated y values does occur, the largest value is taken for the outline of the mountain range.

The algorithms for creating a realistic looking fractal mountain for use within the ISFM system have now been covered. The next chapter explains how these algorithms are actually used with the interactive system.

VI. USE AND IMPLEMENTATION OF THE INTERACTIVE SYSTEM FOR FRACTAL MOUNTAINS

The Interactive System for Fractal Mountains (ISFM) is an interactive system to facilitate the creation of fractal mountains for an observer. ISFM creates the fractal mountains by means of an interface using pop-up menus. Pop-up menus permit the selection of options to change the type of mountain, to change the location of the light source and to create single mountains or mountain ranges. The interactive features permit the observer to see the mountains at different times during the day.

Since the interactive system is controlled by pop-up menus, the menus give the different options that are available. The main menu is:

Fractal Mountains

Tutorial
Texture
Lighting Control
Structural Factor
Create a Mountain
Outline
Create a Mountain Range
Clear and Reset
Exit

Since the system was done to give a systematic approach to the creation of mountains, two types of fractal mountain output have been developed. The first type, the single mountain option, **'Create a Mountain'**, has been included to

give a quick method of reviewing the various types of fractal mountains. The second type, the mountain range option, **'Create a Mountain Range'**, creates a mountain range scene using the various types of fractal mountains. The first four options the **Tutorial, Texture, Lighting Control and Structural Factor** are for use with both a single mountain and a mountain range. The option **Outline** is for use only for creation of a mountain range. Each of the options given above will be discussed in the order they appear.

A. TUTORIAL

The **Tutorial** option gives an online description of the different interactive options available for the creation of fractal mountains. The tutorial on the computer covers the same information as included in this chapter, but is less detail. The tutorial or online information is included so the user can use ISFM directly without reference to this chapter.

B. TEXTURE

Since the different types of mountains and mountain ranges vary in nature, fractal mountains need to be specified by a variety of textures and colors in order to achieve a realistic variety of fractal mountains. The option **Texture** gives five types of defined textures and associated colors. These five types are provided as options from a pop-up submenu of the **Texture** option.

The submenu is:

Texture

rugged granite
rocky mountains
alpine
appalachian
smooth rolling hills

As mentioned in Chapter II.B.4, the ruggedness or smoothness of the texture is determined by a variable called texture. The variable texture determines the range permitted in the displacement of the midpoints. The larger the value of texture, the greater the displacement of the midpoints allowed and the more rugged the appearance of the fractal mountain. The smaller the value of texture, the smaller the displacement of the midpoints and the smoother the appearance of the fractal mountain. The orientation of the base triangles is also given for each option. These orientations are selected to match the type of mountain created. A further discussion of the orientation of the base triangles is found below under the **create a mountain** description. Each of the texture types, along with the color for the mountain, are discussed in the order given in the submenu and are all shown in the summary table, Table 6.1. Sample pictures are given in the results chapter (VII) for all the mountain types.

1. **Rugged Granite**

The **rugged granite** option yields a rock and boulder type texture, The texture variable is set to 15.5. In Chapter V, we state that the surface light properties consist

of ambient light, diffuse light and specular light. These three light types give the surface color for the mountain or mountain range. The typical values selected for the granite surface color are ambient light red = 0.400, green = 0.415 and blue = 0.505. Diffuse light values are red = 0.405, green = 0.405 and blue = 0.500. Specular light values are red = 0.464, green = 0.364 and blue = 0.564. Some specular component is included since a granite surface may be smooth and have directional light reflection. A foreground color is used to give the perception of depth. The color is given with red, green and blue components. The values of these three components range from 0 to 255. The color for granite is red = 175, green = 160 and blue = 178.

The selection of the orientation of the base triangles was made in order to achieve a rugged outline for the mountains. In the single mountain case, the values of z depth range for all the mountains are from -900.0 (close to the eye) to -699.0 (far from the eye). The difference between the different textures appears in the y direction. For the rugged granite, the y values range from 200.0 to 550.0. For the rugged mountain range, the values of z depth range is -1020.0 at y=0 to -700 at y=850.

2. **Rocky Mountains**

The **rocky mountain** option yields a rocky sharp texture. The texture variable is set to 12.5. The surface light properties consist of ambient light, diffuse light and

specular light. The typical values selected for the rocky mountain surface color are ambient light red = 0.565, green = 0.335 and blue = 0.035. Diffuse light values are red = 0.585, green = 0.415 and blue = 0.325. For specular light all three components are set equal to 0.0. The foreground color for rocky mountain is red = 156, green = 102 and blue = 58.

The selection of the orientation of the base triangles is make to achieve a rocky mountain. In the single mountain case, the values of the z depth range is again from -900.0 to -699.0. For the rocky mountain, the y values range from 200 to 500. For the mountain range, the z values range from -1020.0 where y = 0.0 and go to -580.0 where y is 850.0.

3. **Alpine**

The **alpine** option yields a smoother mountain top texture than the rocky mountain option. The texture variable is set to 8.5. The surface light properties consist of ambient light, diffuse light and specular light. The typical values selected for the alpine surface color are ambient light red = 0.665, green = 0.285 and blue = 0.005. Diffuse light values are red = 0.725, green = 0.565 and blue = 0.235. Specular light is set equal to 0.0 for all three components. The foreground color for alpine is red = 201, green = 102 and blue = 58.

The selection of the orientation of the base triangles is made to achieve an alpine mountain. In the single mountain case, the values of the z depth range from -900.0 to -699.0.

For the alpine mountain, the y values range from 200 to 450. For the mountain range, the z values range from -1020.0 where y = 0 and go to -250.0 where y is 850.0.

4. **Appalachian**

The **appalachian** option yields forested texture. The texture variable is set to 5.0. The surface light properties consist of ambient light, diffuse light and specular light. The typical values selected for the appalachian surface color are ambient light red = 0.015, green = 0.465 and blue = 0.025. Diffuse light values are red = 0.105, green = 0.545 and blue = 0.115. The specular light is set equal to 0.0 for all three components. The foreground color for appalachian is red = 61, green = 95 and blue = 36.

The selection of the orientation of the base triangles is make to achieve an appalachian mountain. In the single mountain case, the values of the z depth range from -900.0 to -699.0. For the appalachian mountain, the y values range from 200 to 400. For the mountain range, the z values range from -1020.0 where y = 0 and go to -60 where y is 850.0.

5. **Smooth Rolling Hills**

The **smooth rolling hills** option yields grass texture. The texture variable is set to 2.0. The surface light properties consist of ambient light, diffuse light and specular light. The typical values selected for the smooth rolling hills surface color are ambient light red = 0.025, green = 0.905 and blue = 0.035. Diffuse light values are red

= 0.245, green = 0.994 and blue = 0.155. The specular light is set equal to 0.0 for all three components. The foreground color for smooth rolling hills is red = 0, green = 95 and blue = 40.

The selection of the orientation of the base triangles is make to achieve a rocky mountain. In the single mountain case the values of the z depth range from -900.0 to -699.0. For the smooth rolling hills the y values range from 200 to 300. For the mountain range the z values range from -1020.0 where y = 0 and go to -70 where y is 850.0.

Table 6.1 TEXTURE AND LIGHTING CONDITIONS

	Granite	Rocky	Alpine	Appalachian	Rolling Hills
Texture	15.5	12.5	8.5	5.0	2.0
Ambient					
red	0.400	0.565	0.665	0.015	0.025
green	0.415	0.335	0.285	0.465	0.905
blue	0.505	0.035	0.005	0.025	0.035
Diffuse					
red	0.405	0.585	0.725	0.105	0.245
green	0.405	0.415	0.465	0.545	0.994
blue	0.500	0.325	0.235	0.115	0.155
Specular					
red	0.464	0.0	0.0	0.0	0.0
green	0.364	0.0	0.0	0.0	0.0
blue	0.564	0.0	0.0	0.0	0.0
Foreground					
red	175	156	201	61	0
green	160	102	102	95	95
blue	178	58	58	36	40

C. LIGHTING CONTROL

The lighting is composed of the surface light and color properties, the light source and the light level for the whole scene. In this section, the light source and the light model are discussed. The light source for the fractal mountain is assumed to be the sun. As such, the light source is placed at infinity. Before creating the fractal mountain, pop-up menus give the option to set the time of day.

The menu is specified as:

Lighting Control

Morning
Noon
Evening

Each of the above options are discussed separately and then a summary table of all the values are given in Table 6.2.

1. Morning

For the mountain or mountain range to be viewed in the morning light, the sun location must be in the east and at a low angle from the horizon. The vector x, y, z and w for the light source has values for morning light of x = -1.0, y = 1.5, z = 0.5 and w = 0.0. With the variable w set equal to 0.0, the light source is automatically set at infinity. The other three values determine the light direction. In the case of morning light the numbers indicate that the sun is coming in from the left side of the screen display at about a 45 degree angle from the horizontal negative x axis. It is also

oriented about 40 degrees out of the xy plane. For 40 degrees out from the xy plane the lighting for the tilted base triangles is achieved.

Two other variables determine the light source. These variables are the ambient light and the color of the light. The ambient light values for the morning scene is low because there is not much reflecting light in the morning. The values for ambient light are set to red = 0.2, green = 0.2 and blue = 0.1. The color for the light source is always the same no matter the time of day. The values are set to red = 1.0, green = 0.8 and blue = 0.0.

The last component of the lighting is the light model. The properties in the lighting model are the scene ambient light and the viewer's location. Since the ambient light in the morning is low, the values are set to red = 0.2, green = 0.2 and blue = 0.2. The viewer is placed at infinity, since this is the fastest lighting calculation mode.

2. **Noon**

The next option is the **noon** lighting option. The sun is placed directly behind the observer and at an angle of 60 degrees from the horizontal. This direction of the sun is achieved by the values of x = 0.0, y = 2.0 and z = 1.0. Again w is set equal to 0.0, so the sun is located at infinity. The ambient light from the light source is considered to be higher at noon than in the morning. Values for ambient light are selected as red = 0.6, green = 0.5 and blue = 0.5. The

color of the light source is the same as for morning that is red = 1.0, green = 0.8 and blue = 0.0.

The light of the scene for noon is the same in the morning except for the ambient light. The ambient light for the light source is larger at noon then in the morning. Therefore, the scene ambient light should also be larger. Values for ambient light of the scene are selected as red = 0.6, green = 0.6 and blue = 0.6.

3. **Evening**

The last option is the **evening** light. The values of x, y and z are 1.5, 1.0 and 1.0, respectively. This results is about 30 degrees from the horizontal and about 30 degree angle out from the xy plane. The difference in the direction of the light from the morning compared to the evening is because of the tilt of the base triangles. Also in the morning the light comes from the negative x direction or east and in the evening the light comes from the positive x direction or west.

The ambient light in the evening typically has a red glow, so the value of red is left at 0.6 as in the noon ambient light. The green and the blue values compared to the noon ambient light are decreased to a green value of 0.2 and a blue value of 0.1.

Also, in the evening light was the amount of scene ambient light is changed. The evening ambient light model is more than the morning ambient light, but less than the noon

ambient light. The value for all three colors, red, green and blue, is 0.4.

Table 6.2 LIGHT SOURCE AND SCENE LIGHT FOR LIGHTING CONTROL

Light Properties	Morning	Noon	Evening
Light Source			
Position			
x	-1.0	0.0	1.5
y	1.5	2.0	1.0
z	0.5	1.0	1.3
w	0.0	0.0	0.0
Ambient Light			
red	0.2	0.6	0.4
green	0.2	0.4	0.2
blue	0.1	0.4	0.1
Color			
red	1.0	1.0	1.0
green	0.8	0.8	0.8
blue	0.0	0.0	0.0
Scene Light			
Ambient Light			
red	0.2	0.5	0.3
green	0.2	0.5	0.3
blue	0.2	0.5	0.3
Local Viewer	0.0	0.0	0.0

The result of each of the three light sources and light models is that the triangles with different surface normals are each colored with a different shade. The different shades of color result help in achieving the depth perception of the fractal mountain.

D. STRUCTURAL FACTOR

The **structure factor** option permits variation in the detail of the surface structure. The structure detail is determined by a submenu which gives the options:

Structural Factor

Medium
Fine

1. Medium

When the structural factor is set to **medium,** the recursion stops when the length on an edge (using only the x and z coordinates of the two endpoints) is less than or equal to 16 pixels. If the length of the edge is larger than 16 pixels for any of the three edges of the triangle, then another recursive call is made for that triangle. If the length of all three edges is less than 16, then the triangle is colored, lighted and sent to the screen.

2. Fine

The other option for the structural factor is **fine.** The fine structure causes at least one more recursion level to be produced than the medium structural factor. The number of recursion levels is larger because instead of stopping at the length on an edge of 16 pixels as for the medium structure, the fine structure uses a stopping rule of an edge length of less than or equal to 9 pixels. The result of using

9 pixels for the edge length means a finer detail. The trade off is that it takes more time to generate the fractal mountain.

The medium structure can be used for rugged to alpine mountains. The fine structure should be used for rolling hills. With the fine structure, the triangles used in the construction of the fractal mountain become less noticeable. The problem with using the fine structural factor is that it takes longer to generate the fractal mountain compared to the medium structure.

E. CREATE A MOUNTAIN

The **create a mountain** option is used for creating a single mountain. This option can also be used as a tutorial in the different types of fractal mountains available. In the creation of the mountain, only one base triangle is used. This base triangle is typically tilted out of the xz plane, so that the displacements in the y direction are seen more distinctly by the observer. The tilt is accomplished by placing the triangle on a plane going up in the y direction and going into the screen in the z direction. The amount of change needed in the y direction or in the z direction is determined by the type of mountain desired.

The shape of the tilted base triangle as projected on the xy plane is a factor in determining the type of fractal mountain created. With a base triangle having an acute apex

angle, such as triangle I in Figure 6.1, a sharp peaked mountain is created. If, on the other hand, the point B is changed in the y direction to B' with the same value of x and if the points A and C are kept constant, then the resulting triangle AB'C is closer to an equilateral triangle(see triangle II).

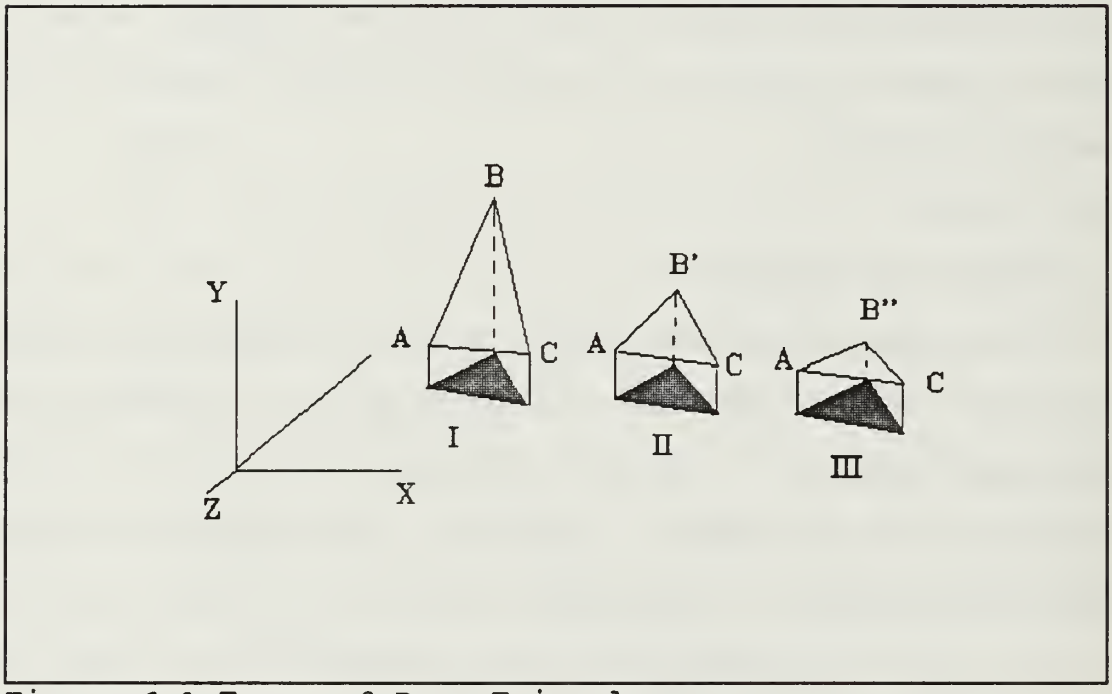

Figure 6.1 Types of Base Triangles

An equilateral triangle gives a smaller or less sharp peak in comparison with the original triangle I. Further decreasing B' to B'' as shown in triangle III results in the triangle AB''C which is an obtuse triangle. The obtuse triangle results in mountains which have smooth or rounded apex angles or tops.

A combination of change in the y direction and changes in the z direction contributes to the type of texture desired in the fractal mountain and the realism of the fractal mountain. To expand the mountain to a more realistic view of nature, the mountain range option is included in ISFM.

F. OUTLINE

The **outline** section is used only with Create a mountain range. The outline allows the observer to create the contour of the mountain range. There is a submenu for drawing the outline which is:

Mountain Outline

create
clear and reset

The main function of the outline section is to place the mountain range wherever the observer wants. The create submenu gives that capability.

1. Create

The **create** option gives a clear screen with the initial point in the center of the window. To create the outline of the mountain range move the cursor along and press the left mouse button. If the cursor moves less than 5 pixels in either the x or y direction than the red line will not be drawn from the previous point to the cursor. The value of 5

is a noise value so that not every slight movement of the cursor is reported. Once the left mouse button is pressed the beginning and ending values of the red line is stored and is then shown as a black line on the screen. The red line now begins at a new location, i.e., the last x and y values. The process of moving the cursor and pressing the left mouse button can continue across the window until the outline is made to fill the whole width of the window. The outline must be a continuous line starting from the left and running to the right or starting from the right and running to the left.

Since the location of the cursor begins in the center of the window and the outline must be continuous, the capability of replacing the original starting point must be available. To replace the starting point or to create a new location for the x and y values the middle mouse button is used. To achieve a new location with the middle mouse button the cursor is moved. This stretches the red line, to the point where the outline is to start. Once the cursor is placed in the desired location, the middle mouse button is depresses and the new location is set. The result is that the red line now starts at this new location. From this new location the rest of the mountain outline can be drawn.

In order to be able to change the outline after having created all or part of the outline, the right mouse button is used. That is, every time the right mouse button is depressed the last point drawn is erased. This erasure of the last

point can be continued until there are all points are erased. This feature permits changing the outline after it has been created.

After the outline has the desired shape for the mountain range, there is the option of exiting the outline. To exit press 'e' on the keyboard. Once the 'e' is pressed the given outline will stay up on the screen as a black contour.

If one wants to change the outline, the create option in the submenu Mountain Outline can be chosen again. If the create option is chosen again the cursor will start at the last point from the previous outline. If on the other hand, one wants to create a completely new outline press the other option clear and reset.

2. **Clear and Reset**

The **clear and reset** option is used to clear the whole window. In this way one obtains an empty window in which a new outline can be created. This option also resets the values in the arrays to zero to start with a reinitialized array. This allows one to create a completely new outline for the mountain range.

G. CREATE A MOUNTAIN RANGE

The **create a mountain range** option is an extension of the **create a mountain** option. Instead of one base triangle, the mountain range is created with multiple base triangles. The

multiple base triangles each have a change in the y direction and a change in the z direction, similar to what was done in the case of one base triangle. The combination of the two changes contributes to the whole surface or texture of the mountain range.

The area in which the mountain range is to be placed is covered with triangles. That is, the triangles make a pattern on the screen display or as a projection on the xy plane, as in Figure 6.2. The triangles are numbered in the order that they create the fractal scene.

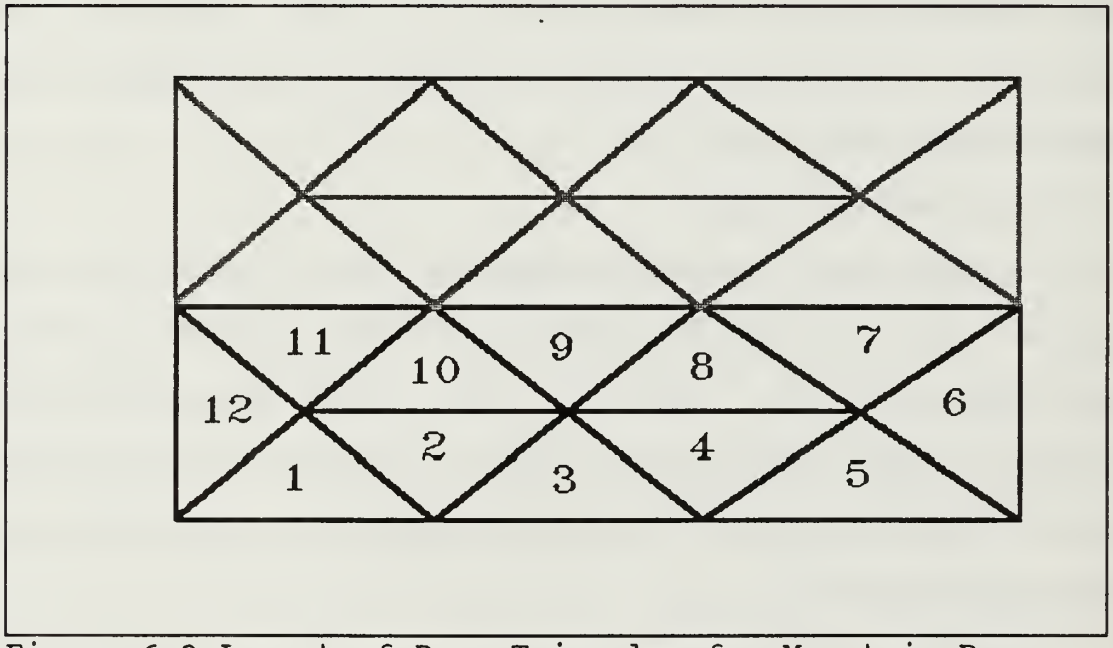

Figure 6.2 Layout of Base Triangles for Mountain Range

In Figure 6.2, the process of creating the mountain range begins in the bottom left hand corner and moves across the row alternating the triangles from one vertex pointing up to one

vertex pointing down. The values assigned to each of the endpoints of adjacent triangles are the same in order to avoid the appearance of the gapping problem. Each vertex of the base triangle pattern is a function of x, y and z coordinates as defined in our usual coordinate system. Each of the triangles 1 through 5, and 7 through 11 are all of the same size. The y coordinates can be varied so that one has triangles with apex angles from obtuse to almost equilateral angles. The size of the triangles is selected to match the type of texture desired. Triangles 6 and 12 are placed on the sides and configured in order to have the size of all the triangles be about the same. The size is important because the number of recursion levels in each triangle depends on the size of the original base triangle. If the number of recursion levels in each base triangle is not approximately the same, then another form of the gapping problem occurs. This gapping problem occurs when considering for the adjacent triangles the midpoint for one triangle is displaced but not for the other adjacent triangle if the number of recursion levels are different. In order to assure that each base triangle uses the same number of recursion levels, the number of recursion levels for the first base triangle is determined based on the size of the smallest triangle desired. For all succeeding base triangles, the stopping criterion is based on the same number of recursion levels instead of the size of the smallest triangles. It is for this reason that the pattern

of covering base triangles has been composed of essentially equal size triangles.

The size of the screen display and the magnitude of the angle of the obtuse triangle used determines the number of total base triangles used to create the entire mountain scene. The pattern of twelve triangles as shown in Figure 6.2 is repeated vertically until the entire screen display is covered.

One of the important elements in creating a mountain range is the depth perception. Depth perception is achieved by using different z values, as described in Chapter V. To provide depth perception to the fractal mountain, the values of z depend on the values of x and y. As the y values increase, the z values also increase. If y was the only coordinate upon which z depended, then depth perception would be directly into the screen. Instead, the z values also depend on the x coordinate. As the x values increase, the z coordinates also increase. The amount of change in the z coordinate is affected more by the y coordinate than by the x coordinate. This enhances the natural look of the scene. With the z coordinate dependent on both x and y coordinates, one has a sense of depth perception and of perspective which is into the screen and towards the right.

The dependence of the z coordinates on both the x and y coordinates helps create a realistic orientation for the

surface of the fractal mountain or mountain range. Variations in the orientation yield different viewing directions.

As explained in the next section color variations and flat surfaces are also introduced in the mountain scene in order to create a further perception of depth.

VII. RESULTS SHOWING FRACTAL MOUNTAINS

ISFM gives the capability of creating various types of mountains. In order to give depth to the pictures, a variation from light to dark color blue was put into the sky. Also a solid foreground polygon is created to give the depth in the pictures. There are a couple other aspects that help the realistic look of the fractal mountains. These aspects are weighted displacement of the midpoints and sophisticated lighting. The weighted displacement of the midpoints allows for less displacement of the midpoint as the triangle size gets smaller. The sophisticated lighting allows the mountains to be see in the morning, at noon and in the evening. For creating the mountains with the options of texture, lighting control and structural factor as well as the aspects just mentioned, one can create a single mountain or a mountain range.

The results of ISFM of each of the sections in Chapter VI is shown by pictures in the following sections. The first option in ISFM is the texture section. The different types of textures are used to show some of the other options available in ISFM. To show the textures from roughness to smoothness, the single mountain case is used. In the single

mountain case, there is a solid foreground corresponding to the color of the type of mountain. The sky color is shaded from a light blue to a shade of dark blue with increasing

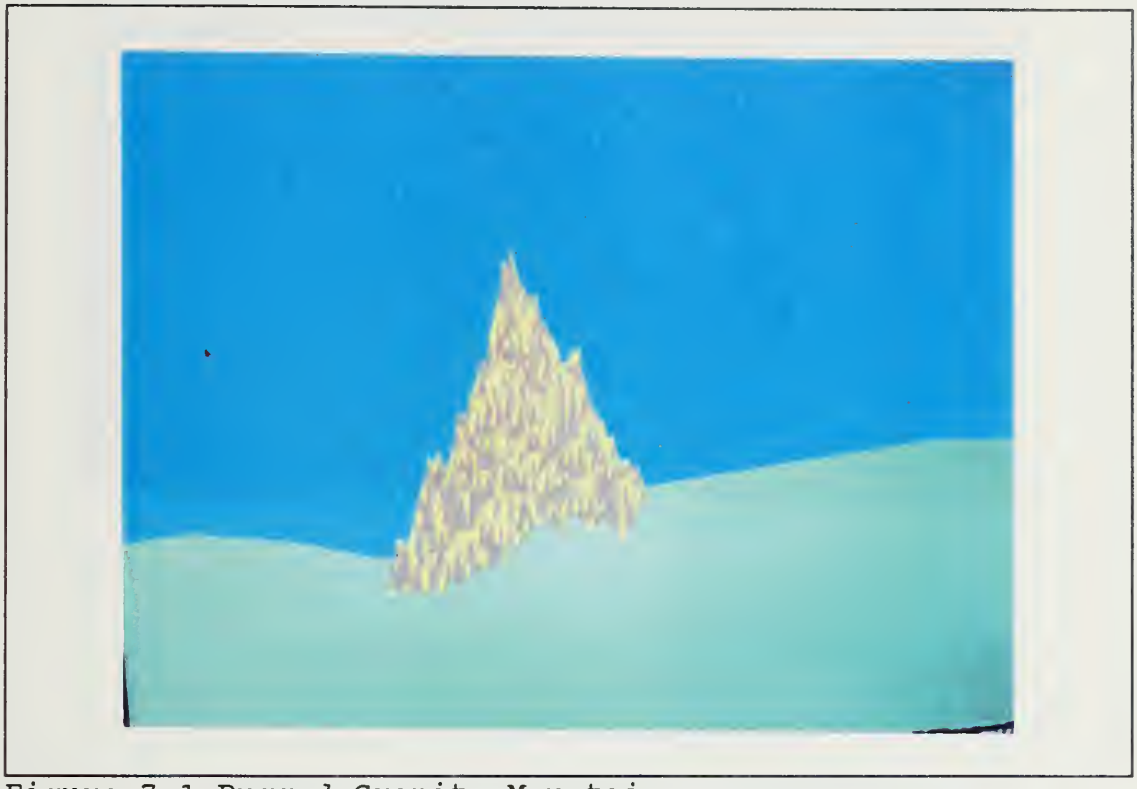

Figure 7.1 Rugged Granite Mountain

height to give the observer the perception of depth. Figure 7.1 gives an example of the most rugged type of texture, that of rugged granite mountain.

The next option of texture, the next step down in roughness, is the texture called rocky mountain. The example of a rocky mountain is shown in Figure 7.2. In comparison to the granite mountain, the texture is different and the mountain is not as high.

Figure 7.2 Rocky Mountain

To show the lighting control options, the texture of alpine mountain has been used as an example. The lighting options are morning, noon and evening. The morning light on an alpine mountain is shown in Figure 7.3. As can be seen the light is coming in from the left side of the observer. Figures 7.4 and 7.5 are examples of the light of noon and evening, respectively, on an alpine mountain. Notice the light on the mountain at noon comes from behind the observer and the light at the evening comes from the right side of the observer.

Figure 7.3 Alpine Mountain - Morning Light

Figure 7.4 Alpine Mountain - Noon Lighting

Figure 7.5 Alpine Mountain - Evening Light

The next option is the structural factor. In order to give an example of the two structural factors, the texture of the appalachian mountain is used. The structural factor of medium is shown in Figure 7.6 and the structural factor of fine is shown in Figure 7.7.

The structure of Figure 7.7 is finer than the structure of Figure 7.6. The finer structure is due to the existence of at least one more recursion level in the mountain formation than with the medium structure option.

Figure 7.6 Appalachian Mountain - Medium

Figure 7.7 Appalachian Mountain - Fine

An option available only in create a mountain is the magnification factor. This magnification factor allows one to see the structure of the mountain in more detail.

The last texture type of smooth rolling hills is used to demonstrate how the magnification looks, Figures 7.8, 7.9, 7.10 and 7.11 have the magnification levels of 1.0, 2.0, 3.0 and 4.0, respectively. These magnifications give the observer a good idea of what the structure actually looks like down to the pixel size.

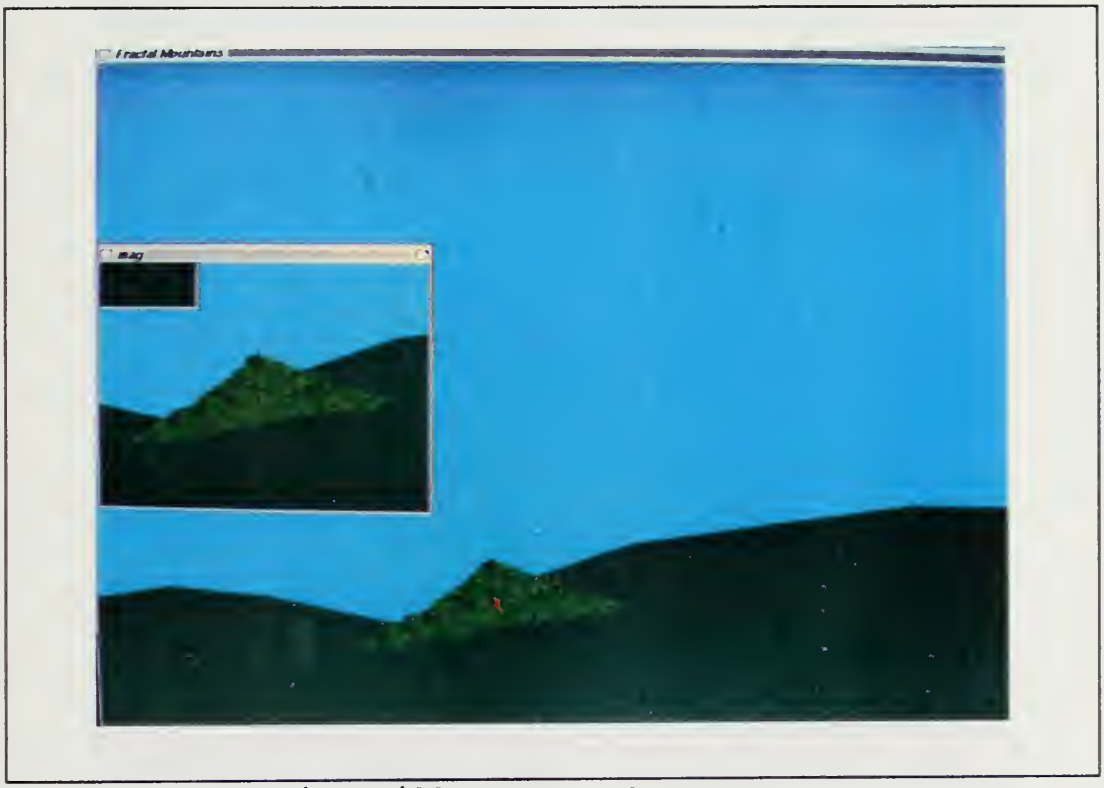

Figure 7.8 Rolling Hills Mag = 1

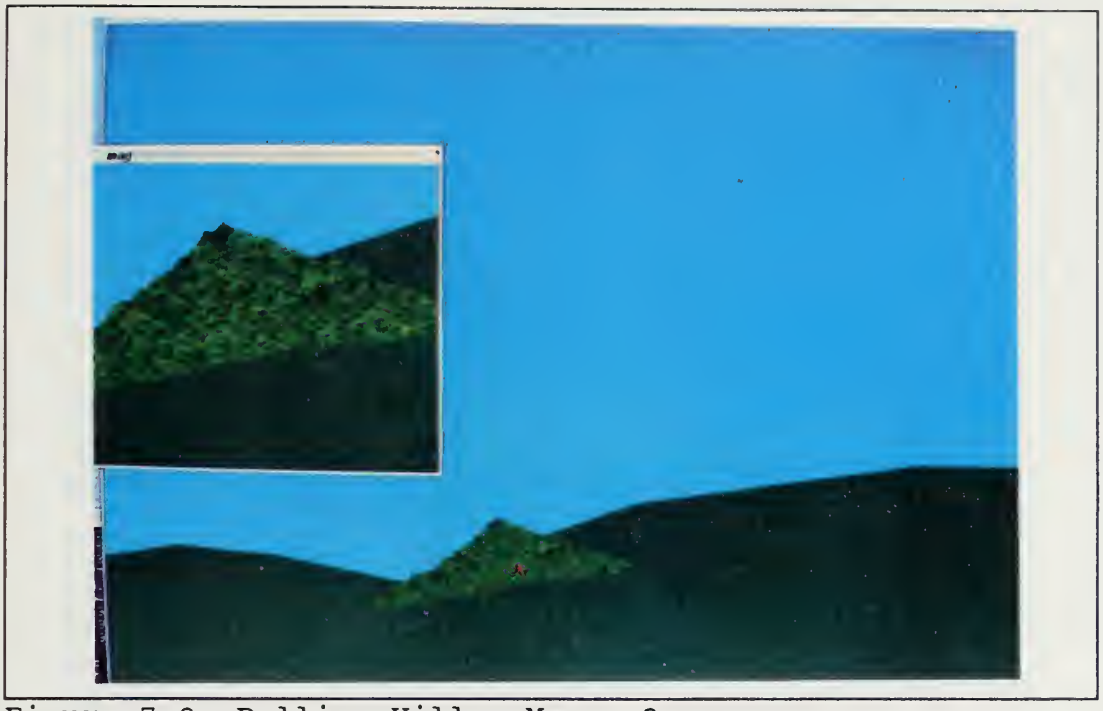

Figure 7.9 Rolling Hills Mag = 2

Figure 7.10 Rolling Hills Mag = 3

Figure 7.11 Rolling Hills Mag = 4

The above figures give all the options available for a single mountain. The other two options that of outline and create a mountain range are shown in a series of pictures of the mountain range given for different outlines. The outline of each mountain range is indicative of the type of texture of the mountain range. The texture again is given from the most rough to the smoothest. Each of the options of lighting control and structural factor is incorporated in one or more of the examples. Figure 7.12 is an example of a rugged granite mountain range with a morning light and a structural factor of medium.

Figure 7.12 Rugged Granite Mountain Range - Morning Light - Medium

The next example, Figure 7.13, is of a rocky mountain range with a noon lighting an a medium structural factor.

Figure 7.14 is an example of an alpine mountain range with evening light and a medium structure factor.

To show an example of an appalachian mountain, the options of morning light and fine structure factor are shown in Figure 7.15.

The last mountain range texture is the smooth rolling hills, as seen in Figure 7.16. The smooth rolling hills picture has the options of an evening light and a fine structural factor.

Figure 7.13 Rocky Mountain Range - Noon Lighting - Medium

Figure 7.14 Alpine Range - Evening Light - Medium

Figure 7.15 Appalachian Range - Morning Light - Fine

Figure 7.16 Rolling Hills - Evening Light - Fine

The above pictures Figures 7.1- 7.16 give the available options for both creating a mountain and creating a mountain range. These figures are a representation of the different possibilities available while using ISFM. The results display a good deal of realism and natural effects typical of mountain scenery.

VIII LIMITATIONS, FUTURE DIRECTIONS AND CONCLUSION

The Interactive System for Fractal Mountain (ISFM) is user friendly and facilitates the creation of Fractal Mountains. It has the capability of creating various type of mountains. In order to give depth to the pictures, a variation from light to dark color blue was imparted to the sky. Also a solid foreground polygon is created to lend the depth to the pictures. The use of sophisticated lighting routines and depth perception is enhanced by coloring and flat space segments in the mountains and the mountain range.

The Interactive System for Fractal Mountain (ISFM) can be expanded to include seasonal effects for the mountain ranges. The seasonal effects can be accomplished by varying the basic grid pattern for the mountain range. Other things that could be done with the system, are to introduce other natural aspects such as lakes and rivers within the scene. Along with including lakes and rivers, a whole mountain scene could be included by a progression of different textures across the basic grid pattern. Different textures within a mountain scene would allow the creation of rolling hills and rocky mountains to be in the same picture. To permit the different textures within the same picture to be chosen by the user, the user would need to be able to work on different grid sections

of the mountains independently. Working on different sections could also allow the possibility of color variation across the different sections of the picture. This would allow, for example, the creation of snow-capped mountains by the user.

Two other possible extensions to the interactive system concern the outline and the foreground. The outline should match the type of mountain being created. This could be done by randomly displacing the points of the given outline in the same manner as used in the displacement for the mountain texture. Another useful addition to the interactive system would be the ability to vary the size and shape of the foreground contour. This variation would give different depth perceptions to the user.

Not only can this interactive system be expanded to include more capabilities for the fractal mountains, but also the system could be used to create other natural objects like clouds, trees and ocean surfaces. These other natural objects could use some of the same techniques used in creating fractal mountains. One of the possible applications could be based on different base shapes. In this study, a triangle is used as the base shape. To create other natural objects, shapes such as squares, rectangles or any other polygon could be used.

The Interactive System for Fractal Mountain (ISFM) is a straightforward method of developing fractal mountains. ISFM gives the capability of creating various types of mountains.

ISFM is only the first part of a project whose ultimate goal is the complete description of the techniques necessary for utilizing fractals for the generation of realistic texturing for computer generated images.

APPENDIX : C PROGRAMS

PROGRAM : MNTMAIN.C

```
/*  This is program mntmain.c
    sets up the pop-up menus for Fractal mountains */

#include "gl.h"
#include "device.h"
#include "list.h"
#include "math.h"
#include "stdio.h"

#define EXIT 99

 /* integer variables for menus - main, mountain outline,
    mountain texture, lighting control, structural factor,
    magnification and exit                                  */

int mainmenu;
int mntmenu;
int txtrmenu;
int lsmenu;
int strtmenu;
int exmenu;
int crtmenu;

 /* integer variables denoting the current texture, structural
    factor and light position                               */

int curlsor;
int curtxtr;
int strtfct;

 /* variables denoting the current magnification factor
    texture and single mountain or mountain range       */
int imag;
float texture;
int m;

/* Counters  */
int IC,Tri;
int I1;

 /* pointer variables for link list which stores the vertices  */
LINK first,frst,tail;
float SR;

 /*  array to store the vertex values                       */
float AS[10];
 /* array for the random numbers */
double RAND[500];
```

```
 /* arrays for storing the outline and integers for total amount
   of points in each array                                    */
float pts[1000][3];
int totops;
float alpts[1000][3];
int ntot;

long op[1000];              /* The operation to perform for the outline  */

 /* flags for mountain outline, number of levels, logical variable
    if 1 or more base triangles have been created, default outline used
    and how many triangle patterns used in creating mountain range  */
int flg;
int nlev,ct;
int otflg;
float squfac;


 /* function to set the light source at morning, noon or evening  */
   lsp(n)
  {
    curlsor = n;
    return 3;
  }

 /* function to set the texture type rugged granite to smooth rolling hills */
  txtrmnt(n)

  int n;
  {
   curtxtr = n;
   return 2;
  }

 /* function to set Structural factor medium or fine  */
  strtmnt(n)

  int n;
  {
   strtfct = n;
   return 4;
  }

 /* function to set the magnification of the single mountain  */
  crtmnt(n)

  int n;
  {
   imag = n;
   return 5;
  }
```

```
main()
{
/* value for reading from the queue  */
short value;
int i;

/* integer value for pop-up menus    */
static int pupval = 1;

/* create and open Fractal Mountains Window   */
prefsize(1050,850);

winopen("Fractal Mountains");
wintitle("Fractal Mountains");
winconstraints();

/* Place in the modes singlebuffer and RGB
   and set the window depth and screen coordinates  */

singlebuffer();
RGBmode();
viewport(0,1050,0,850);
lsetdepth(0x0000,0x7fffff);
ortho(0.0,1050.0,0.0,850.0,1023.0,0.0);
gconfig();
qdevice(REDRAW);
qdevice(MENUBUTTON);

/* Build the random number table  */
rand_table_gen();

RGBcolor(0,175,255);     /* color blue*/
clear();

/* create the intro window with text    */
  RGBcolor(0,0,0);
  cmov2i(275,600);
  charstr("Welcome to Interactive System for Fractal Mountains(ISFM)");
  cmov2i(250,450);
  charstr("If this is your first time please see the Tutorial section");
  cmov2i(250,400);
  charstr("for a description of the options and defaults");
  cmov2i(200,150);
  charstr("Program designed and written by Patricia Collins-Hunt");

/* Create the Second level exit menu                */
 exmenu = defpup("Exit %t |Yes I am sure %x99|No don't do it");

/* Create the menu for Position of the Light Source                */
 lsmenu = defpup("Time of Day %t %F |morning %x31|noon %x32 ",lsp);
 addtopup(lsmenu, "evening %x33");
```

```
/* Create the menu for mountain outline                */
 mntmenu = defpup("Mountain Outline %t");
 addtopup(mntmenu, "create %x61");
 addtopup(mntmenu, "clear screen %x62");

/* Create the menu for Magnification                   */
 crtmenu = defpup(" Magnification  %t %F|1.0 %x1 |2.0 %x2 |3.0 %x3",crtmnt);
 addtopup(crtmenu, "4.0 %x4");

/* Create the menu for mountain texture                */

 txtrmenu = defpup("Mountain Texture %t %F | rugged-granite %x21",txtrmnt);
 addtopup(txtrmenu,"rocky mountains %x22 |alpine %x23 |appalachian %x24" );
 addtopup(txtrmenu,"smooth rolling hills %x25" );

/* Create the menu for Structural Factor               */
 strtmenu = defpup("Structural Factor %t %F | medium %x16",strtmnt);
 addtopup(strtmenu,"fine %x9" );

/* Create the main menu for Fractal Mountains          */
 mainmenu = defpup("Fractal Mountains %t |Tutorial %x1");
 addtopup(mainmenu,"Texture %m",txtrmenu);
 addtopup(mainmenu,"Lighting Control %m",lsmenu);
 addtopup(mainmenu,"Structural Factor %m",strtmenu);
 addtopup(mainmenu,"Create a Mountain %m ", crtmenu);
 addtopup(mainmenu,"Outline %m ",mntmenu);
 addtopup(mainmenu,"Create a Mountain Range %x7");
 addtopup(mainmenu,"clear and reset %x8|exit %x97");

 /* initialize the variables denoting the current options  */
 curtxtr = 0;
 curlsor = 0;
 strtfct = 0;

 /* Go into loop for pop-up menus until exit is chosen  */
 flg = 0;
 while(pupval != EXIT)
 {
   switch(qread(&value))
   {
    case REDRAW:
      reshapeviewport();
      break;

   case MENUBUTTON:
    if (value == 1)
    {
      pupval = dopup(mainmenu);
      switch(pupval)
      {
       case 1:          /* for Tutorial   */
        tutor1();
        break;
```

```
      case 5:           /* create a mountain   */
       m=1;
       sampmnt();
       break;
      case 7:          /* create a mountain range  */
       m=2;
       sampmnt();
       break;
      case 8:         /* clear and reset option */
       RGBcolor(0,175,255);
       clear();
       break;
      case 61:       /* for creating outline for mountain range */
       drawmnt();
       break;
      case 62:       /* for clear and reset outline option     */
       RGBcolor(0,175,255);
       clear();

       for (i=0;i<=1000;i++)
        {
          alpts[i][0]=0.0;
          alpts[i][1]=0.0;
          alpts[i][2]=0.0;

          pts[i][0]=0.0;
          pts[i][1]=0.0;
          pts[i][2]=0.0;
          flg = 0;
        }
       break;
      case 97:               /* for exit option    */
       pupval = dopup(exmenu);
       break;
    }
   }
   break;
  default:
   break;
  }
 }
}
```

PROGRAM : MOUNTDYN.C

```
/*
     This program initializes the system to light and color the fractal
     mountain or the mountain range  */


/*  The variables are the following :
    texture-gives the value of the roughness of the mountain
    IC-a counter for how many times went through gendyn
    Tri - how many triangles are created
    first,frst,tail - pointers to addresses of the array of
       storing the vertices
    AS[10]- stores the three endpoints of the triangles and puts
       them into dynamic storage
    ysno- variable to send triangle to screen
    curtxtr - value of current texture of options
    curlsor - value of current light position of options
    m - determines if single mountain or mountain range
    squfac - how many sets of 12 triangles in pattern for
       mountain range
    P1,P2,P3 - endpoints of the triangles                    */

#include <gl.h>
#include <device.h>
#include <math.h>     /* Standard UNIX include file for math library  */
#include "list.h"
#include "stdio.h"
#include "lightdefs.h"


#define x 0
#define y 1
#define z 2

/* Global Tables */

   extern float texture;
   extern int IC,Tri;
   extern LINK first,frst,tail;
   extern float AS[10];
   extern float ysno;
   extern int curtxtr;
   extern int curlsor;
   extern int m;
   extern float squfac;
   extern float magfac;


  /* BEGIN SAMPMNT */

    sampmnt()
```

```
{
  /* Local Variables */

     float P1[3],P2[3],P3[3];

     Colorindex wmask;

  /*Enable all bitplanes for writing*/
     wmask=(1<<getplanes())-1;
     writemask(wmask);

     wintitle("Mountain Scene");
     mmode(MVIEWING);
     RGBmode();
     singlebuffer();
     drawmode(NORMALDRAW);
     gconfig();



  /* Fractalize until desired precision */
   IC = 0;
   Tri = 0;

/* initialize the lighting values  */
     initialmat();
     initialls();
     initiallm();

     zbuffer(TRUE);
     zclear();


    /*  determine the type of texture */

   if (curtxtr == 0) curtxtr = 23;

   switch(curtxtr)
     {
      case 21:          /* rugged granite  */
        lmbind(MATERIAL,MATGRANITE);
        texture = 15.5;
        squfac = 2.0;
        break;

      case 22:          /*  rocky mountains  */
        lmbind(MATERIAL,MATROCKS);
        texture = 12.5;
        squfac = 2.0;
        break;

      case 23:          /*  alpine  mountains */
        lmbind(MATERIAL,MATSUNSET);
        texture = 8.5;
        squfac = 3.0;
        break;
```

```
        case 24:          /* appalacian mountains  */
           lmbind(MATERIAL,MATFOREST);
           texture = 5.0;
           squfac = 4.0;
           break;

         case 25:           /* smooth rolling hills  */
           lmbind(MATERIAL,MATGRASS);
           texture = 2.0;
           squfac = 5.0;
           break;

          }

/* determine location of the light source  */

      if (curlsor == 0) curlsor = 31;
      switch(curlsor)
        {
         case 31:           /* morning sun  */
           lmbind(LMODEL,LMODM);
           lmbind(LIGHT0,LSOURCM);
           break;

         case 32:           /*  noon sun      */
           lmbind(LMODEL,LMODN);
           lmbind(LIGHT0,LSOURCN);
           break;

         case 33:           /*  evening sun */
           lmbind(LMODEL,LMODE);
           lmbind(LIGHT2,LSOURCE);
           break;
         }

/*  detrmine if a mountain or a mountain range is desired  */
      switch(m)

        {
         case 1:              /* for creating a mountain   */
           mountcal();
           break;

         case 2:             /* for creating a mountain range */
           interpmnt();
           basetri();
           break;

        }

         zbuffer(FALSE);

      wintitle("Fractal Mountains");

    /* END mountdyn */
   }
```

PROGRAM : MOUNTCAL.C

```
/*   This program initializes the basetriangle for creating a mountain */

/* The external variables are the following:
    I1,IC, - counters to determine number of base triangles
    Tri - number of triangles created
    nlev - total number of recursive levels
    ct - counter for determining more than one base triangle
    curtxtr - value from options of current texture
    imag - amount of magnification
                                                              */
#include <math.h>     /* Standard UNIX include file for math library  */
#include "list.h"
#include "stdio.h"


#define x 0
#define y 1
#define z 2

/* Global Tables */

   extern int I1,IC,Tri;
   extern LINK first,frst,tail;
   extern int nlev,ct;
   extern int curtxtr;
   extern int imag;


   mountcal()
      {
        /* Local Variables */

        /* variables for the endpoints of the triangles   */
           float P1[3],P2[3],P3[3];
           int    ilev;

        /* variables for foreground polygon               */
           float frdl[3],frdr[3],frul[3],frur[3]forg1[3];
           float forg2[3],forg3[3],forg3a[3],forg4[3],forg5[3],forg6[3];

        /* variables for sky polygon                      */
           float skdl[3],skdr[3],skul[3],skur[3];
           float cup[3],cdn[3];

        /* initialize values for foreground polygon */
           frdl[0]=0.0;
           frdl[1]=0.0;
           frdl[2]= -300.0;

           frdr[0]=1050.0;
           frdr[1]=0.0;
           frdr[2]= -300.0;
```

```
        frul[0]=0.0;
        frul[1]=250.0;
        frul[2]= -300.0;

        frur[0]=1050.0;
        frur[1]=355.0;
        frur[2]= -300.0;

        forg1[0]=85.0;
        forg1[1]=265.0;
        forg1[2]= -300.0;

        forg2[0]=200.0;
        forg2[1]=257.0;
        forg2[2]= -300.0;

        forg3[0]=355.0;
        forg3[1]=240.0;
        forg3[2]= -300.0;

        forg3a[0]=470.0;
        forg3a[1]=297.0;
        forg3a[2]= -300.0;
        forg4[0]=600.0;
        forg4[1]=320.0;
        forg4[2]= -300.0;

        forg5[0]=745.0;
        forg5[1]=340.0;
        forg5[2]= -300.0;

        forg6[0]=915.0;
        forg6[1]=360.0;
        forg6[2]= -300.0;

/* Create the initiating structure  for the mountain */
      switch(curtxtr)
       {
       case 21:             /* rugged granite case  */
         P1[x] = 300.0;
         P1[y] = 200.0;
         P1[z] = -900.0;

         P2[x] = 450.0;
         P2[y] = 550.0;
         P2[z] = -699.0;

         P3[x] = 600.0;
         P3[y] = 300.0;
         P3[z] = -800.0;

         RGBcolor(136,143,128);
       break;
```

```
case 22:            /* rocky mountain case  */
  P1[x] = 300.0;
  P1[y] = 200.0;
  P1[z] = -900.0;

  P2[x] = 450.0;
  P2[y] = 500.0;
  P2[z] = -699.0;

  P3[x] = 600.0;
  P3[y] = 300.0;
  P3[z] = -800.0;

  RGBcolor(75,42,16);
break;

case 23:            /* alpine case  */
  P1[x] = 300.0;
  P1[y] = 200.0;
  P1[z] = -900.0;

  P2[x] = 450.0;
  P2[y] = 450.0;
  P2[z] = -699.0;

  P3[x] = 600.0;
  P3[y] = 300.0;
  P3[z] = -800.0;

  RGBcolor(159,88,69);
break;

case 24:            /* appalachian case  */
  P1[x] = 300.0;
  P1[y] = 200.0;
  P1[z] = -900.0;

  P2[x] = 450.0;
  P2[y] = 400.0;
  P2[z] = -699.0;

  P3[x] = 600.0;
  P3[y] = 300.0;
  P3[z] = -800.0;

  RGBcolor(61,115,36);
break;

case 25:            /* smooth rolling hills case  */
  P1[x] = 300.0;
  P1[y] = 200.0;
  P1[z] = -900.0;
```

```
      P2[x] = 450.0;
      P2[y] = 300.0;
      P2[z] = -699.0;

      P3[x] = 600.0;
      P3[y] = 250.0;
      P3[z] = -800.0;

      RGBcolor(0,95,40);
    break;

   }


/* initialize counters and flag   */
    ilev = 0;
    I1 = 0;
    IC = 0;
    Tri = 0;
    ct = 0;

   /* create the foreground structure   */

    bgnpolygon();
      v3f(frdl);
      v3f(frul);
      v3f(forg1);
      v3f(forg2);
      v3f(forg3);
      v3f(forg3a);
      v3f(forg4);
      v3f(forg5);
      v3f(forg6);
      v3f(frur);
      v3f(frdr);
    endpolygon();


   /* Initialize the sky structure  */

    skdl[0] = 0.0;
    skdl[1] = 200.0;
    skdl[2] = -100.0;

    skdr[0] = 1050.0;
    skdr[1] = 200.0;
    skdr[2] = -100.0;

    skul[0] = 0.0;
    skul[1] = 850.0;
    skul[2] = -100.0;

    skur[0] = 1050.0;
    skur[1] = 850.0;
    skur[2] = -100.0;
```

```
    cup[0] = 0.0;
    cup[1] = 0.386;
    cup[2] = 1.0;

    cdn[0] = 0.0;
    cdn[1] = 0.88;
    cdn[2] = 1.0;

   /* create the sky structure  */

    bgnpolygon();
      c3f(cdn);
      v3f(skdl);
      v3f(skdr);
      c3f(cup);
      v3f(skur);
      v3f(skul);
    endpolygon();


  /* create the fractal mountain   */
        mountain_generate(P1,P2,P3,ilev);

        listlite(first);

system("mag imag");

  ct = 1;

    return;
/* END mountcal */
}
```

```
PROGRAM : RANDTBL.C

/*   This program tests the random number generation for the
     mountain fractal gaussian distribution */


#include <math.h>     /* Standard UNIX include file for math library  */

/* external global variables */

extern double RAND[500];


  /* BEGIN RAND_TABLE_GEN */

    rand_table_gen()

      {
        /* Local Variables */

           int    I,J;
           double UNF1, UNF2;
           double range,pi;
           int    factor;

           pi = 3.1415926535;

        /* Determine the range for the random numbers of UNIX UCB */

           range = 2;
           for (J=1; J<=14; J++)
              {
                 range = range * 2;
              }
           range = range - 1;

        /* Set the random number generator seed */

           srand(475836);

        /* Create a Table for 500 entries */

           for (I=0; I< 500; I = I + 2)
              {
               /* Get a uniform random number through the Unix C subroutine */

                  UNF1 = rand();
                  UNF2 = rand();
```

```
        /* Normalize the uniform random number to the interval [0,1] */

           UNF1 = UNF1 / range;
           UNF2 = UNF2 / range;

        /* Mold the uniform random variable into the approximate normal
           distribution */

           factor = 1.0;
           if (log(UNF1) < 0.0) factor = -1.0;
           RAND[I] = sqrt(factor * (2.0 * log(UNF1))) *
                          cos ((2*pi*UNF2));
           RAND[I] = RAND[I] * factor;
           factor = 1.0;
           if (log(UNF2) < 0.0) factor = -1.0;
           RAND[I+1] = sqrt(factor * (2.0 * log(UNF1))) *
                          sin ((2*pi*UNF2));
           RAND[I+1] = RAND[I+1] * factor;
        }
        return;
}
```

PROGRAM : LIST.H

```
/*  This file is used by other programs in creating the dynamic
  variable. In this program we create the linked list to be store
  the points*/

#define NULL 0

typedef float AP;

struct linked_list
 {
    float AP;
    struct linked_list   *prev;
    struct linked_list   *next;
 };

 typedef struct linked_list   ELEMENT;
 typedef ELEMENT              *LINK;
```

PROGRAM : LISTLTD.C

```
 /* This routine will pick up the stored values and then
    light the polygons  */

/*  The variables are the following:
     IC, I1 are counters for number of times gendyn and listlite used
     frst,tail,lst  - pointers for addresses in dynamic storage
     AS[10] - array for storing endpoints of the triangles in
         dynamic storage
                                                          */
#include "list.h"
#include "stdio.h"
#include <math.h>

extern int IC,I1;
extern LINK frst,tail;
extern float AS[10];

   listlite(head)

  LINK head;

   {
   LINK lst;
   int i,j;


     IC=IC+1;
     lst = head;
     if(IC==1) frst = head;
     if(IC==1) lst = tail;

  /* retrieve the three coordinates of each point and a flag    */
    while (lst != NULL)
     {
       for(i=9;i>=0;i=i-1)
        {
       AS[i] = lst->AP;
       free(lst);
       lst = lst-> prev;
        }


     /* send endpoints to litemnt to light the triangles and send
        to the screen                                        */
      litemnt();
     }
    return;
     }
```

PROGRAM : BASETRI.C

```
/* This routine will pick up the stored values and then
    light the polygons  */

#define x 0
#define y 1
#define z 2

#include "gl.h"
#include "device.h"
#include "list.h"
#include "stdio.h"
#include "lightdefs.h"
#include "math.h"

/*  the variables are the following:
    pts and alpts - arrays for values of a given outline
    totops - total # of points in alpts array
    ntot - total # of points in plts array
    RAND - array for the random variabales
    texture - value for each of the different textures
    first, tail and frst - pointers to addresses of the array for
       storing the vertices
    I1,IC - counters for new base triangles
    Tri - counts # of triangles created
    nlev - total # of recursion levels
    ct - flag for determining if one or more base triangles created
    squfac - how many time triangle pattern is repeated for
       mountain range
    curtxtr - current texture value from menu
    otflg - for default outline
                                                                */

extern float pts[1000][3];
extern int totops;
extern double RAND[500];
extern float texture;
extern LINK first,tail,frst;
extern int I1,IC,Tri;
extern float alpts[1000][3];
extern int ntot;
extern int nlev,ct;
extern float squfac;
extern int curtxtr;
extern int otflg;
```

```
basetri()

{

/* the variables are the following :
    ilev - the current recursion level
    cont - vector of points to create each vertex of a polygon
    i, j, k - loop counters for pattern of mountain range
    fg - counter for the current base triangle in the pattern
    maxtri - maximum number of base triangles created

    vupr, vupl, vdownr, vdownl, vnxtr, vnxtl - arrays for creating
        the sky polygon
    cvd, cvup - color arrays for the sky polygon
    ts - variable to make sure outline starts at 0 and ends at 1023
    vn - 1 or -1 for creating each series of 12 triangles
    yfac - a factor used in the pattern for the mountain range
    P1, P2, P3 - values of endpoints of the triangle
    TP - temporary array holding values of the endpoints
    loc - variable to determine location in array RAND
    frmdl, frmdr, frmul, frmur - arrays for creating foreground polygon
    frm1 - frm6 - arrays for creating foreground polygon
    znn, zp, zn, zpp - variables for changing z values for each texture

                                                                    */


  int ilev;
  float cont[3];
  int i,j,k;
  int fg,maxtri;
  float vupr[3],vupl[3],vdownr[3],vdownl[3];
  float vnxtr[3],vnxtl[3];
  float cvd[3],cvup[3];
  float ts,vn;
  float yfac;
  float P1[3],P2[3],P3[3],TP[3];
  int loc;
  float frmdl[3],frmdr[3],frmul[3],frmur[3];
  float frm1[3],frm2[3],frm3[3],frm4[3],frm5[3],frm6[3];
  float znn,zp,zn,zpp;

  winconstraints();

/*  initialize values for the sky polygon   */

  vupr[0]=1050.0;
  vupr[1]=1050.0;
  vupr[2]= -1023.0;

  vupl[0]=0.0;
  vupl[1]=1050.0;
  vupl[2]= -1023.0;
```

```
    vdownr[0]=1050.0;
    vdownr[1]=0.0;
    vdownr[2]= -1023.0;

    vdownl[0]=0.0;
    vdownl[1]=0.0;
    vdownl[2]= -1023.0;

    vnxtr[0]=1050.0;
    vnxtr[1]=10.0;
    vnxtr[2]= -1023.0;

    vnxtl[0]=0.0;
    vnxtl[1]=100.0;
    vnxtl[2]= -1023.0;

    cvd[0] = 0.0;
    cvd[1] = 0.88;
    cvd[2] = 1.0;

    cvup[0] = 0.0;
    cvup[1] = 0.386;
    cvup[2] = 1.0;

/* determine if need the default outline or not                */
    if (otflg == 0) defout();
    loc = 0;

    RGBcolor(255,255,255);
    clear();

    /* create sky polygon  */
    bgnpolygon();

     for(j=0;j<=ntot;j=j+1)
      {
       cont[x]=(alpts[j][x]);
       cont[y]=(alpts[j][y]);
       cont[z]= -1023.0;
       cvd[1] = ((cont[y]* -0.00058)+0.880);
       c3f(cvd);
       v3f(cont);
      }

   /*  create first and last vertices                          */

     ts= (alpts[ntot][x]);
     if(ts <=525) cont[x]=0.0;
     if(ts > 525) cont[x]=1050.0;

     cont[y]=(alpts[ntot][y]);
     cont[z] = -1023.0;
     c3f(cvd);
     v3f(cont);
```

```
    if (cont[x] == 0.0)
        {
         c3f(cvup);
         v3f(vupl);
         c3f(cvup);
         v3f(vupr);
         if (alpts[0][x] != 1050.0)
          {
           cont[x] =1050.0;
           cont[y] = (alpts[0][y]);
           cont[z] = -1023.0;
           c3f(cvd);
           v3f(cont);
           }
        }

        else
         {
         c3f(cvup);
         v3f(vupr);
         c3f(cvup);
         v3f(vupl);
         if (alpts[0][x] != 0.0)
          {
           cont[x] =0.0;
           cont[y] = (alpts[0][y]);
           cont[z] = -1023.0;
           c3f(cvd);
           v3f(cont);
           }
         }
     endpolygon();

/* initialize the foreground for the mountain range  */

  frmur[0]=1050.0;
  frmur[1]=150.0;
  frmur[2]= -1023.0;

  frmul[0]=0.0;
  frmul[1]=40.0;
  frmul[2]= -1023.0;

  frmdr[0]=1050.0;
  frmdr[1]=0.0;
  frmdr[2]= -1023.0;

  frmdl[0]=0.0;
  frmdl[1]=0.0;
  frmdl[2]= -1023.0;

  frml[0]=200.0;
  frml[1]=43.0;
  frml[2]= -1023.0;
```

```
  frm2[0]=400.0;
  frm2[1]=45.0;
  frm2[2]= -1023.0;

  frm3[0]=530.0;
  frm3[1]=65.0;
  frm3[2]= -1023.0;

  frm4[0]=645.0;
  frm4[1]=82.0;
  frm4[2]= -1023.0;

  frm5[0]=790.0;
  frm5[1]=105.0;
  frm5[2]= -1023.0;

  frm6[0]=930.0;
  frm6[1]=130.0;
  frm6[2]= -1023.0;

/* vary the change in the z coordinates for each type of texture  */

   switch (curtxtr)
    {

    case 21:            /* for rugged granite  */
     RGBcolor(136,143,128);
     zp = 20.0;
     zn = 10.0;
     zpp = 40.0;
     znn = 210.0;
     break;

    case 22:           /* for rocky mountain  */
     RGBcolor(75,42,16);
     zp = 30.0;
     zn = 20.0;
     zpp = 60.0;
     znn = 220.0;
     break;

    case 23:           /* for alpine mountain */
     RGBcolor(159,88,69);
     zp = 40.0;
     zn = 25.0;
     zpp = 65.0;
     znn = 230.0;
     break;

    case 24:          /* for appalachian mountain */
     RGBcolor(61,95,36);
     zp = 50.0;
     zn = 25.0;
     zpp = 55.0;
     znn = 240.0;
     break;
```

```
   case 25:          /* for rolling hills  */
    RGBcolor(0,95,40);
    zp = 40.0;
    zn = 30.0;
    zpp = 40.0;
    znn = 190.0;
    break;
   }
/* create foreground  */

  bgnpolygon();
   v3f(frmdl);
   v3f(frmul);
   v3f(frm1);
   v3f(frm2);
   v3f(frm3);
   v3f(frm4);
   v3f(frm5);
   v3f(frm6);
   v3f(frmur);
   v3f(frmdr);
  endpolygon();

/* initialize variables for fractal mountain  range */
  yfac = (800.0/squfac)/2.0;
 I1=0;
 IC=0;
 Tri = 0;
 ct = 0;

/* create the first base triangle   */
  P1[x] = 0.0;
  P1[y] = 0.0;
  P1[z] = -1020.0;

  P2[x] = 175.0;
  P2[y] = yfac;
  P2[z] = -860.0;

  P3[x] = 350.0;
  P3[y] = 0.0;
  P3[z] = -940.0;

 k=0;
 fg=1;
ilev = 1;

/* start creation of fractal mountain range  */

 mountain_generate(P1,P2,P3,ilev);
 mntrange(first);

 ct = 1;
 vn=1.0;
```

```
*determine the maximum number of  base triangles to create      */
 maxtri= (squfac * 12)-1;

 for(j=1;j<=maxtri;j=j+1)
 {

 I1=0;
 IC=0;

 fg=fg+1;

/* create triangles 1-5, 7-11 in pattern   */
  if (fg != 6 && fg != 12)
   (
    P1[x] = P2[x];
    P1[y] = P2[y];
    P1[z] = P2[z];

    P2[x] = P3[x];
    P2[y] = P3[y]+((2*yfac)*k);
    P2[z] = P3[z]+(100.0*k);

    P3[x] = P1[x]+(350.0*vn);
    P3[y] = P1[y];

/* change the depth of the z value for each texture  */
    if (P3[y] > P2[y])
      {
       P3[z] = P1[z]+zp;
      }
    else
      {
       P3[z] = P1[z]+zn;
      }

    if (fg > 6 && fg < 12)
    (
    if (P3[y] < P2[y])
      (
       P3[z] = P1[z]-zp;
      }
    else
      {
       P3[z] = P1[z]-zn;
      }
    }

    if (k >= 1) k=0;
   }


 /* create triangle 6 in the pattern for mountain range */
  if (fg == 6)
   (

    vn= -1.0;
```

```
      P1[x] = P3[x];
      P1[y] = P3[y];
      P1[z] = P3[z];

      P3[x] = P2[x];
      P3[y] = P2[y];
      P3[z] = P2[z];

      P2[x] = P1[x];
      P2[y] = P1[y]+(yfac*2.0);
      P2[z] = P2[z]+zpp;

     }

   /* create triangle 12 in the pattern for mountain range */
    if (fg == 12)
     {

     P1[x] = P3[x];
     P1[y] = P3[y]-(yfac*2.0);
     P1[z] = P3[z]-znn;

     TP[x] = P2[x];
     TP[y] = P2[y];
     TP[z] = P2[z];

     P2[x] = P3[x];
     P2[y] = P3[y];
     P2[z] = P3[z];

     P3[x] = TP[x];
     P3[y] = TP[y];
     P3[z] = TP[z];

     znn = 100.0;
     vn=1.0;
     k= k+1;
     fg=0;

     }

   ilev = 1;

/* send the values to create the fractal  */
   mountain_generate(P1,P2,P3,ilev);
   mntrange(first);

   }

   return;
     }
```

PROGRAM : DRAWMNT.C

```
/* this function draws the mountain range outline

   The left mouse draws a line and adds to accumulative object.
   The middle mouse moves to a new position.
   The program is exited by popup menu on the right mouse.

*/

#include "gl.h"
#include "device.h"

/* define the world coordinate boundaries */

#define WXMIN 0.0
#define WXMAX 1050.0
#define WYMIN 0.0
#define WYMAX 850.5
#define WZMIN 1023.0
#define WZMAX 0.0

#define MOVETO 11  /* opcodes for our complete picture */
#define DRAWTO 21
#define DELDRAW 31

#define MAXPTS 1000  /* max number of points */
extern float pts[MAXPTS][3];   /* the xyz coords */
extern long totops;            /* total number of points in pts  */
extern float alpts[MAXPTS][3];  /* editted xyz coordinates to use for
                                   outline of mountain range       */
extern int ntot;            /* total number of points in alpts    */
extern int flg;             /* flag to determine if new outline or not */
extern int otflg;         /* determines if default outline is used    */
extern long op[MAXPTS];     /* the operation to perform       */

drawmnt()
{
   /* the single line's location... */
   float single[2][3];

   /* temp wx,wy coords of line. */
   float wx,wy,wz;

   /* value returns from the event queue */
   short value,valuex,valuey;

   /* temp variable */
   float junk;

   long i;   /* loop temp */
```

```
/* device name returned from the event queue */
short temp;
int EX;                    /*used to exit outline section  */
float ver[3];             /* array used to create outline   */
float tpx,tpy,tp;    /* variables used to check for duplicate points */
int j,k;             /*loop variables   */

/* put up some help text as the title */
wintitle("Mouse values - Draw / New Location / Erase /    Exit-KEYBD e
   line must be
/*  continuous and drawn left to right    */
 /* make button hits on the mouse go to the event queue */
 qdevice(RIGHTMOUSE);
 qdevice(MIDDLEMOUSE);
 qdevice(LEFTMOUSE);
 qdevice(KEYBD);

 /* queue mouse dial movements */
 qdevice(MOUSEX);
 qdevice(MOUSEY);

 /* only report movements of 5 or more to the queue */
 noise(MOUSEX,5);
 noise(MOUSEY,5);

 /* if middle button is hit, report mousex + y */
 tie(MIDDLEMOUSE,MOUSEX,MOUSEY);

 /* if Left button  is hit, report mousex + y */
 tie(LEFTMOUSE,MOUSEX,MOUSEY);

 /* queue the REDRAW event (i.e. reshape has been called from the
    window manager)                                              */

 qdevice(REDRAW);

 /* attach the cursor to the mouse */
 attachcursor(MOUSEX,MOUSEY);

 /* turn the cursor on */
 curson();
 zbuffer(TRUE);

/* determine if new or continued outline                    */
 if (flg ==1)
  {
   wx = pts[totops][0];
   wy = pts[totops][1];
   wz = -1023.0;
  }
 else
   {
   /* initial location for the cursor */
   wx = WXMIN + (WXMAX - WXMIN)/2.0;
   wy = WYMIN + (WYMAX - WYMIN)/2.0;
   wz = -1023.0;
   }
```

```
mmode(MSINGLE);
gconfig();

/* set the mouse to be at the world coordinate spot wx,wy */
newmouselocation(wx,wy);
/* set up the initial line */
single[0][0] = wx;
single[0][1] = wy;
single[0][2] = wz;

single[1][0] = wx;
single[1][1] = wy;
single[1][2] = wz;

/* put a single moveto into the overall picture */
totops = totops + 1;
op[totops] = MOVETO;
pts[totops][0] = wx;
pts[totops][1] = wy;
pts[totops][2] = wz;


EX = 2;

/* wait for the window manager to exit us...*/
while(EX != 10)
{

   /* while we have values on the Q, process them */
   while(qtest() != 0)
   {

      switch(qread(&value))
      {
         case KEYBD:
                     if(value == 101) /*e*/
                     {
                     /*totops = totops -1;*/

                     /* make the single line back one point  */
                     single[0][0] = pts[totops][0];
                     single[0][1] = pts[totops][1];
                     single[0][2] = pts[totops][2];
                     single[1][0] = pts[totops][0];
                     single[1][1] = pts[totops][1];
                     single[1][2] = pts[totops][2];

                     /* set new point to the previous point */
                     op[totops+1] = op[totops];
                     pts[totops+1][0]=pts[totops][0];
                     pts[totops+1][1]=pts[totops][1];
                     pts[totops+1][2]=pts[totops][2];

                      EX = 10;
                     }
                     break;
```

```
case REDRAW: /* should we redraw the picture? */
             reshapeviewport();

             /* put the mouse in its new location */
             newmouselocation(wx,wy);
             break;

case MIDDLEMOUSE: /* if middle MOUSE, start a new line */

             /* next 2 records on Q are mousex and mousey */
             temp=qread(&valuex);
             temp=qread(&valuey);

             /* compute the world coordinate */
             computeworldcoord(valuex,valuey,&wx,&wy);

             /* make the single line be here */
             single[0][0] = wx;
             single[0][1] = wy;
             single[0][2] = wz;
             single[1][0] = wx;
             single[1][1] = wy;
             single[1][2] = wz;

             /* put a moveto into the big picture */
             totops = totops + 1;
             op[totops] = MOVETO;
             pts[totops][0] = wx;
             pts[totops][1] = wy;
             pts[totops][2] = wz;

             break;

case LEFTMOUSE: /* if left MOUSE, draw the line */

             /* read the queue for the mousex + y */
             temp=qread(&valuex);
             temp=qread(&valuey);

             /* compute the world coordinate of the mouse */
             computeworldcoord(valuex,valuey,&wx,&wy);

             /* make the single line be here */
             single[0][0] = wx; single[0][1] = wy;
             single[0][2] = wz;
             single[1][0] = wx;
             single[1][1] = wy;
             single[1][2] = wz;

             /* put a drawto into the big picture */
             totops = totops + 1;
             op[totops] = DRAWTO;
             pts[totops][0] = wx;
             pts[totops][1] = wy;
             pts[totops][2] = wz;

             break;
```

```
        case RIGHTMOUSE:     /* If right mouse delete last point */

                    totops = totops -1;

                    /* make the single line back one point  */
                    single[0][0] = pts[totops][0];
                    single[0][1] = pts[totops][1];
                    single[0][2] = pts[totops][2];
                    single[1][0] = pts[totops][0];
                    single[1][1] = pts[totops][1];
                    single[1][2] = pts[totops][2];

                    op[totops+1] = op[totops];
                    pts[totops+1][0]=pts[totops][0];
                    pts[totops+1][1]=pts[totops][1];
                    pts[totops+1][2]=pts[totops][2];

                    break;

        case MOUSEX: /* we have movement on the mouse in the X direction */

                    /* get the world coordinates */
                    computeworldcoord(value,valuey,&wx,&junk);

                    /* must change the x coord of the single line */
                    single[1][0] = wx;
                    break;

        case MOUSEY: /* we have movement in the Y direction on the mouse */

                    /* get the world coordinates */
                    computeworldcoord(valuex,value,&junk,&wy);

                    /* must change the y coord of the single line */
                    single[1][1] = wy;
                    break;

        default:
                    break;

    } /* end switch statement */

}   /* end while there is stuff on the Q (qtest() != 0)  */


/* rebuild the complete picture */
RGBcolor(0,175,255);
clear();

/* draw the lines thicker */
 linewidth(2);

/* draw the big picture in black */
RGBcolor(0,0,0);
```

```
    /* draw the big picture */
    for(i=0; i <= totops; i=i+1)
    {
      bgnline();
       switch(op[i])
       {
          case MOVETO:
                      endline();
                      bgnline();
                      ver[0]=pts[i][0];
                      ver[1]=pts[i][1];
                      ver[2]=pts[i][2];
                      v3f(ver);
                      break;
          case DRAWTO:
                      ver[0]=pts[i][0];
                      ver[1]=pts[i][1];
                      ver[2]=pts[i][2];
                      v3f(ver);
                      break;
          default:
                      break;
       }
    }
    endline();

    /* draw the single line */
    RGBcolor(255,0,0);
    bgnline();
    for (i=0;i<=1;i=i+1)
      {
      ver[0]=single[i][0];
      ver[1]=single[i][1];
      ver[2]=single[i][2];
      v3f(ver);
      }

    endline();

}

/* create a new array without duplicate point and starting
   with the third point.                                    */

 j=0;
   alpts[j][0] = pts[j+2][0];
   alpts[j][1] = pts[j+2][1];
   alpts[j][2] =  -1023.0;

  for(k=3;k<=totops;k++)
  {
    tpx = abs(pts[k][0]-alpts[j][0]);
    tpy = abs(pts[k][1]-alpts[j][1]);
    tp = tpx;
    if (tpy > tpx) tp = tpy;
```

```
        if (tp >= 5)
         {
         j=j+1;
         alpts[j][0] = pts[k][0];
         alpts[j][1] = pts[k][1];
         alpts[j][2] =  -1023.0;
         }
      }
   ntot = j;
   if (ntot <= 3) otflg = 0;

    wintitle("Fractal Mountains");
    zbuffer(FALSE);

    flg = 1;
}
/* put the mouse where the world coordinate last was */

newmouselocation(wx,wy)

float wx,wy;   /* last position in world coords for the mouse */

{
   long originx,originy;   /* origin of the graphics port */

   long sizex,sizey;       /* size of the graphics port */

   long initx,inity;       /* where the mouse should be in the graphics port */


   /* re-get the location and size of the graphics port */
   getorigin(&originx,&originy);
   getsize(&sizex,&sizey);

   /* we must reset the mouse cursor to be at the location
      of the last x + y coordinate set */
   initx = ((sizex - 1)/(WXMAX - WXMIN)) * (wx - WXMIN) + originx;
   inity = ((sizey - 1)/(WYMAX - WYMIN)) * (wy - WYMIN) + originy;

   setvaluator(MOUSEX,initx,0,XMAXSCREEN);
   setvaluator(MOUSEY,inity,0,YMAXSCREEN);

}
/* compute the world coordinate corresponding to this mouse position */

computeworldcoord(mousex,mousey,wx,wy)

short mousex,mousey;   /* the location of the mouse */

float *wx,*wy;         /* the returned world coord of the mouse */
```

```
{
   long originx,originy;    /* origin of the graphics port */

   long sizex,sizey;        /* size of the graphics port */


   /* re-get the location and size of the graphics port */
   getorigin(&originx,&originy);
   getsize(&sizex,&sizey);

   /* compute the world coordinate */
   *wx = ((WXMAX - WXMIN)/(sizex - 1)) * (mousex - originx) + WXMIN;
   *wy = ((WYMAX - WYMIN)/(sizey - 1)) * (mousey - originy) + WYMIN;

}
```

PROGRAM GENDYN.C

```
/* This is file gendyn.c that generates the fractals  */
#include "list.h"
#include "stdio.h"
#include <math.h>

#define x 0
#define y 1
#define z 2

/* Global Structures */

/* The variables are the following:
    RAND - array for the random variables
    texture - value for each of the different textures
    I1,IC - counters for number of base triangles
    Tri - counts # of triangles created
    tail, first -pointers to address of dynamic array
    strtfct - current structural factor
    nlev - total number of recursion levels
    ct - flag for determining if one or more base triangles created

                                                                    */
extern double RAND[500];
extern float texture;
extern int I1,IC,Tri;
extern LINK tail,first;
extern int strtfct;
extern int nlev,ct;


  /* BEGIN RECURSIVE PROCESS */

      mountain_generate(P1,P2,P3,ilev)

            /* Parameter variables */
            /* P1,P2,P3 -values of endpoint of the triangles
               ilev - current recursion level                   */

               float P1[3],P2[3],P3[3];
               int    ilev;

      {

         /* Local variables
             I - loop variable
             Pmid1,Pmid2,Pmid3 - midpoints of x, y and z coordinates
             TEMP1,TEMP2, TEMP3 - distance squared between endpoints
                of x and z coordinates
             TEMP - smallest distance squared value between endpoints
             DIST - Square root of TEMP
```

```
        TWO - is equal to 2.0
        DPAR - array for putting endpoints into linked_list
        yn - flag to determine if triangle small enough
        PTX,PTZ - variables used to determine location in RAND
        rhead - value returned from ray_to_list
        PT - value to mod to get location in RAND
        loc - location in RAND                                    */

        int    I;
        float Pmid1[3],Pmid2[3],Pmid3[3];
        float TEMP,DIST,TWO;
        float TEMP1,TEMP2,TEMP0;
        float DPAR[10];
        float yn;
        float PTX,PTZ;
        /* *ray_to_list is a function which returns a pointer to a float
           variable */
        LINK ray_to_list();
        LINK rhead;
        float PT;
        int loc;


        TWO = 2.0;
 /* find the Distance of x and z coordinates of the two endpoints
    of each edge                          */

       TEMP0 = (P2[x]-P1[x])*(P2[x]-P1[x])+
               (P2[z]-P1[z])*(P2[z]-P1[z]);
       TEMP = TEMP0;
       TEMP1 = (P3[x]-P2[x])*(P3[x]-P2[x])+
               (P3[z]-P2[z])*(P3[z]-P2[z]);
       if (TEMP1 > TEMP ) TEMP = TEMP1;
       TEMP2 = (P1[x]-P3[x])*(P1[x]-P3[x])+
               (P1[z]-P3[z])*(P1[z]-P3[z]);
       if (TEMP2 > TEMP ) TEMP = TEMP2;

       DIST = sqrt(TEMP);

 if(strtfct == 0) strtfct = 16;
    I1=I1+1;
if (ct == 0)
{
if (DIST < strtfct)
    {
    yn = 1.0;

     for(I=0;I<=2;I++)
       {
       DPAR[I]=P1[I];
       DPAR[I+3]=P2[I];
       DPAR[I+6]=P3[I];
       }
       DPAR[9]=yn;
```

```
        if (I1==1) rhead = ray_to_list(DPAR);
        if (I1>1)ray_next(DPAR);
        return;
        }
    }

/* for the mountain range if it is not the first base triangle stop
   iterations at the same level as the first base triangle   */

    if (ct == 1)
    {
    if (ilev == nlev)
        {
        yn = 1.0;

         for(I=0;I<=2;I++)
           {
           DPAR[I]=P1[I];
           DPAR[I+3]=P2[I];
           DPAR[I+6]=P3[I];
           }
           DPAR[9]=yn;

        if (I1==1) rhead = ray_to_list(DPAR);
        if (I1>1)ray_next(DPAR);
        return;
        }
    }

  /* find the midpoints of x, y and z coordinates   */


          for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
             {
              Pmid1[I] = (P1[I] + P2[I]) / TWO;
             }

          for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
             {
              Pmid2[I] = (P2[I] + P3[I]) / TWO;
             }

          for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
             {
              Pmid3[I] = (P3[I] + P1[I]) / TWO;
             }

    /* Displace  randomly the midpoints of the y coordinate   */

       /* for the edge between first and second endpoints  */
            PTX=(P1[x]+P2[x]);
            PTZ=(-P1[z]+ -P2[z]);
            PT = PTZ + PTX;
            loc = (long)(PT) % 500L;
```

```
        if (TEMP0 < 50.0)
         {
        Pmid1[y] = ((TEMP0/50.0)*(texture * RAND[loc])) + Pmid1[y];
         }
        else
         {
        Pmid1[y] = (texture * RAND[loc]) + Pmid1[y];
         }

 /* for the edge between second and third endpoints   */
        PTX=(P2[x]+P3[x]);
        PTZ=( -P2[z]+ -P3[z]);
        PT = PTZ + PTX;
        loc = (long)(PT) % 500L;
        if (TEMP1 < 50.0)
         {
        Pmid2[y] = ((TEMP1/50.0)*(texture * RAND[loc])) + Pmid2[y];
         }
        else
         {
        Pmid2[y] = (texture * RAND[loc]) + Pmid2[y];
         }

 /* for the edge between third and first endpoints   */
        PTX=(P3[x]+P1[x]);
        PTZ=( -P3[z]+ -P1[z]);
        PT = PTZ + PTX;
        loc = (long)(PT) % 500L;
        if (TEMP2 < 50.0)
         {
        Pmid3[y] = ((TEMP2/50.0)*(texture * RAND[loc])) + Pmid3[y];
         }
        else
         {
        Pmid3[y] = (texture * RAND[loc]) + Pmid3[y];
         }

 yn = 0.0;

for(I=0;I<=2;I++)
   {
   DPAR[I]=P1[I];
   DPAR[I+3]=P2[I];
   DPAR[I+6]=P3[I];
   }
    DPAR[9]=yn;
if (I1==1) rhead = ray_to_list(DPAR);
if (I1==1)first=rhead;

ilev = ilev + 1;
if (I1>1)ray_next(DPAR);
```

```
        /* Recurse on the triangles to create smaller triangles */

                Tri = Tri + 1;
                mountain_generate(Pmid1,P2,    Pmid2,ilev);
                Tri = Tri + 1;
                mountain_generate(P3,    Pmid3,Pmid2,ilev);
                Tri = Tri + 1;
                mountain_generate(Pmid1,Pmid3,P1    ,ilev);
                Tri = Tri + 1;

                if (ct == 0) nlev = ilev;
                mountain_generate(Pmid1,Pmid3,Pmid2,ilev);
        /* END generate */
           return;
        }
```

PROGRAM : RAYLT.C

```
/* This file raylt.c dynamicly stores the first set of points
      for creating the mountains.
      It transfers the arrays into a list.*/

#include "list.h"
#include <math.h>

extern LINK tail;
LINK ray_to_list(DPAR)

  float DPAR[];
  {
     LINK head=NULL;
     LINK oldst;
     int i;
     if(DPAR[0]>='\0')   /*base case*/
      {
      head = (LINK)malloc(sizeof(ELEMENT));
      head ->AP = DPAR[0];
      head ->prev = NULL;

       tail = head;
    /* store the rest of the 9 values of the DPAR values    */
      for(i=1;i <= 9;i++)
       {
         tail ->next = (LINK)malloc(sizeof(ELEMENT));
         oldst = tail;
         tail = tail -> next;
         tail -> AP =DPAR[i];
         tail -> prev = oldst;

       }
      return(head);
      }
  }
```

PROGRAM : RAYNT.C

```
/*This file raynt is to dynamicly store the points for creating
the mountains after the first triangle is stored with raylt.c*/

#include "list.h"
extern LINK tail;
extern int IC;

  ray_next(DPAR)

   float DPAR[];
    {
      int i,ki;
      LINK oldst;
        for(i=0;i <= 9;i++)
         {
          tail -> next= (LINK)malloc(sizeof(ELEMENT));
          oldst = tail;
          tail = tail -> next;
          tail -> AP=DPAR[i];
          tail -> prev = oldst;
         }
      return;
      }
```

PROGRAM : INTERPMNT.C

```
/* This routine interprets between the points and then takes care
    of backtracking and intersections  */

#define x 0
#define y 1

#include "list.h"
#include "stdio.h"
#include <math.h>

/* The variables are the following:
   IC,I1 - counters for  new base triangles
   alpts - array for values of a given outline
   ntot - total number of points in alpts array
                                                    */
extern int IC,I1;
extern float alpts[1000][3];
extern int ntot;

   interpmnt()

   {
   /* Local Variables
     i, k - loop variables
     j - variable to increment from 0 to 1049
     ix - last value of each interpolation set
     kl - variable used for interpolation
     intout - array to hold interpolated values
     sl - slope between any two points
     it - variable used in replacing the largest point into
        array alpts
                                                          */

   int i,j,ix,k,kl;
   int intout[1050][2];
   float sl;
   int it;


   for (i=0;i<=1049;i=i+1)
     {
       intout[i][x] = i;
       intout[i][y] = 0;
     }

  /* start at the far lefthand side                */
   j = 0;
   ix = (int)(alpts[j][x]);

  /*  if the outline started on the left side use this if statement */
   if (ix < 525)
    {
```

```
/* create intout array from the x values of point 1 to point 2 of array
   alpts */
 intout[ix][y] = (int)(alpts[j][y]);
 intout[j][y] = (int)(alpts[j][y] - 25.0);
 sl = (alpts[j][y] - (float)(intout[j][y]))/(float)(ix - j);

/* interpolate between points 1 and 2                              */
 kl=0;
 for (k=j+1;k<=ix-1;k=k+1)
   {
   kl=kl+1;
    intout[k][y] = (int)(sl*kl) + intout[j][y];
   }

/*create the rest of the y values of intout array */
 for (i=0;i<=ntot;i++)
   {
   j=ix;
   ix = (int)(alpts[i+1][x]);

/* if the x values of the two points are equal take the largest
   value of y of the two points                                    */
    if (ix == j)
     {
       if ((int)(alpts[i+1][y]) > intout[ix][y])
        {
          intout[ix][y]=(int)(alpts[i+1][y]);
        }
     }



/* if the values of the second point in x is smaller than the first
   then check the y values and get the largest                     */
     if (ix < j)
      {



   /* if the y value of the second is larger than what is already
       there recompute the y values between the x coordinates of
       the second point and the point previous                     */

       if ((int)(alpts[i+1][y]) > intout[ix][y])
        {
          intout[ix][y]=(int)(alpts[i+1][y]);
          sl = (alpts[i+1][y]-alpts[i-1][y])/(alpts[i+1][x]-alpts[i-1][x]);
          j = (int)(alpts[i-1][x]);

       /* do interpolation                                         */
          for (k=j+1;k<ix;k=k+1)
           {
            kl=kl+1;
            intout[k][y] = (int)(sl*kl) + intout[j][y];
           }
        }
```

```
   /* if the value is smaller reset the second x point and go to the
      next point in alpts                                          */

        else
        {
        ix = (int)(alpts[i-1][x]);
        }
      }

   /* if second of the two points is larger                        */
    if (ix > j)
     {
     intout [ix][y] = (int)(alpts[i+1][y]);
     sl = (alpts[i+1][y]-alpts[i][y])/(float)(ix-j);

       /* do interpolation                                        */
     kl=0;
     for (k=j+1;k<ix;k=k+1)
      {
       kl=kl+1;
       intout[k][y] = (int)(sl*kl) + intout[j][y];
      }
     }
    }

  /* do from the last point in array alpts to the far right 1049 */
    j = ix;
    ix = 1049;
    intout[ix][x] = ix;
    intout[ix][y] = (int)(alpts[ntot][y] + 25.0);
    sl = (25.0/(float)(ix-j));

       /* do interpolation                                        */
    kl=0;
    for (k=j+1;k<ix;k=k+1)
      {
       kl=kl+1;
       intout[k][y] = (int)(sl*kl) + intout[j][y];
      }
    }

/* replace the values in alpts with the value in intout
    if the value in the array alpts is smaller than
    the value in intout                                          */
  for (i=0;i<=ntot;i++)
    {

    it = (int)(alpts[i][x]);

    if (intout[it][y] > (int)(alpts[i][y]))
     {
      alpts[i][y] = (float)(intout[it][y]);
     }
    }

  return;
  }
```

PROGRAM : MNTRANGE.C

```
/* This routine will pick up the stored values and then
    light the polygons  for a mountain range */

#include "list.h"
#include "stdio.h"
#include <math.h>

/* The variables are the following:
   IC,I1 - are counters for number of times subroutines gendyn and listlite
      are used
   frst,tail,lst  - pointers for addresses in dynamic storage
   AS[10] - array for storing endpoints of the triangles in
      dynamic storage                                          */

extern int IC,I1;
extern LINK frst,tail;
extern float AS[10];

   mntrange(head)

  LINK head;

   {
   LINK lst;
   int i,j;      /* loop variables  */


     IC=IC+1;
     lst = head;
     if(IC==1) frst = head;
     if(IC==1) lst = tail;
    while (lst != NULL)
     {
       for(i=9;i>=0;i=i-1)
        {
       AS[i] = lst->AP;
       free(lst);
       lst = lst-> prev;
        }

      litemnt();
     }
    return;
     }
```

```
PROGRAM : DEFOUT.C

 *   This program initializes the outline for creating a mountain */


#include <math.h>     /* Standard UNIX include file for math library  */
#include "list.h"
#include "stdio.h"


#define x 0
#define y 1
#define z 2

/* Global Tables */
/* The variables are the following:
    curtxtr - current texture value from the menu
    alpts - array for values of a given outline
    ntot - total number of points in alpts array
    i - counter for current point
                                                        */
   extern int curtxtr;
   extern float alpts[1000][3];
   extern int ntot;


   defout()
      {
        /* Local Variables */
       int i;

    switch(curtxtr)
     {
     case 21:     /* rugged granite   */
      i=0;
      i = i+1;
      alpts[i][x] = 0.0;
      alpts[i][y] = 250.0;
      i = i+1;
      alpts[i][x] = 100.0;
      alpts[i][y] = 325.0;
      i = i+1;
      alpts[i][x] = 260.0;
      alpts[i][y] = 550.0;
      i = i+1;
      alpts[i][x] = 330.0;
      alpts[i][y] = 450.0;
      i = i+1;
      alpts[i][x] = 502.0;
      alpts[i][y] = 750.0;
      i = i+1;
      alpts[i][x] = 675.0;
      alpts[i][y] = 345.0;
      i = i+1;
      alpts[i][x] = 900.0;
```

```
 alpts[i][y] = 625.0;
 i = i+1;
 alpts[i][x] = 1050.0;
 alpts[i][y] = 400.0;
 ntot = 8;
break;

case 22:      /* rocky mountain   */
 i=0;
 i = i+1;
 alpts[i][x] = 0.0;
 alpts[i][y] = 300.0;
 i = i+1;
 alpts[i][x] = 155.0;
 alpts[i][y] = 550.0;
 i = i+1;
 alpts[i][x] = 210.0;
 alpts[i][y] = 500.0;
 i = i+1;
 alpts[i][x] = 380.0;
 alpts[i][y] = 700.0;
 i = i+1;
 alpts[i][x] = 450.0;
 alpts[i][y] = 580.0;
 i = i+1;
 alpts[i][x] = 560.0;
 alpts[i][y] = 445.0;
 i = i+1;
 alpts[i][x] = 746.0;
 alpts[i][y] = 375.0;
 i = i+1;
 alpts[i][x] = 850.0;
 alpts[i][y] = 500.0;
 i = i+1;
 alpts[i][x] = 945.0;
 alpts[i][y] = 670.0;
 i = i+1;
 alpts[i][x] = 1000.0;
 alpts[i][y] = 600.0;
 i = i+1;
 alpts[i][x] = 1050.0;
 alpts[i][y] = 515.0;
 ntot = 11;
break;

case 23:      /* alpine mountain   */
 i=0;
 i = i+1;
 alpts[i][x] = 0.0;
 alpts[i][y] = 350.0;
 i = i+1;
 alpts[i][x] = 100.0;
 alpts[i][y] = 425.0;
 i = i+1;
```

```
 alpts[i][x] = 150.0;
 alpts[i][y] = 460.0;
 i = i+1;
 alpts[i][x] = 250.0;
 alpts[i][y] = 570.0;
 i = i+1;
 alpts[i][x] = 300.0;
 alpts[i][y] = 530.0;
 i = i+1;
 alpts[i][x] = 395.0;
 alpts[i][y] = 375.0;
 i = i+1;
 alpts[i][x] = 465.0;
 alpts[i][y] = 395.0;
 i = i+1;
 alpts[i][x] = 600.0;
 alpts[i][y] = 455.0;
 i = i+1;
 alpts[i][x] = 725.0;
 alpts[i][y] = 575.0;
 i = i+1;
 alpts[i][x] = 775.0;
 alpts[i][y] = 654.0;
 i = i+1;
 alpts[i][x] = 990.0;
 alpts[i][y] = 375.0;
 i = i+1;
 alpts[i][x] = 1050.0;
 alpts[i][y] = 405.0;
 ntot = 12;
break;

case 24:     /* applachian mountain   */
 i=0;
 i = i+1;
 alpts[i][x] = 0.0;
 alpts[i][y] = 275.0;
 i = i+1;
 alpts[i][x] = 75.0;
 alpts[i][y] = 305.0;
 i = i+1;
 alpts[i][x] = 125.0;
 alpts[i][y] = 325.0;
 i = i+1;
 alpts[i][x] = 200.0;
 alpts[i][y] = 355.0;
 i = i+1;
 alpts[i][x] = 275.0;
 alpts[i][y] = 360.0;
 i = i+1;
 alpts[i][x] = 350.0;
 alpts[i][y] = 325.0;
 i = i+1;
 alpts[i][x] = 425.0;
 alpts[i][y] = 335.0;
```

```
 i = i+1;
 alpts[i][x] = 500.0;
 alpts[i][y] = 355.0;
 i = i+1;
 alpts[i][x] = 575.0;
 alpts[i][y] = 405.0;
 i = i+1;
 alpts[i][x] = 650.0;
 alpts[i][y] = 425.0;
 i = i+1;
 alpts[i][x] = 725.0;
 alpts[i][y] = 420.0;
 i = i+1;
 alpts[i][x] = 800.0;
 alpts[i][y] = 385.0;
 i = i+1;
 alpts[i][x] = 875.0;
 alpts[i][y] = 365.0;
 i = i+1;
 alpts[i][x] = 925.0;
 alpts[i][y] = 340.0;
 i = i+1;
 alpts[i][x] = 1000.0;
 alpts[i][y] = 325.0;
 i = i+1;
 alpts[i][x] = 1050.0;
 alpts[i][y] = 330.0;
 ntot = 16;
break;

case 25:     /* smooth rolling hills   */
 i=0;
 i = i+1;
 alpts[i][x] = 0.0;
 alpts[i][y] = 200.0;
 i = i+1;
 alpts[i][x] = 75.0;
 alpts[i][y] = 225.0;
 i = i+1;
 alpts[i][x] = 125.0;
 alpts[i][y] = 245.0;
 i = i+1;
 alpts[i][x] = 200.0;
 alpts[i][y] = 265.0;
 i = i+1;
 alpts[i][x] = 275.0;
 alpts[i][y] = 274.0;
 i = i+1;
 alpts[i][x] = 350.0;
 alpts[i][y] = 255.0;
 i = i+1;
 alpts[i][x] = 425.0;
 alpts[i][y] = 240.0;
```

```
 i = i+1;
 alpts[i][x] = 500.0;
 alpts[i][y] = 230.0;
 i = i+1;
 alpts[i][x] = 575.0;
 alpts[i][y] = 245.0;
 i = i+1;
 alpts[i][x] = 650.0;
 alpts[i][y] = 257.0;
 i = i+1;
 alpts[i][x] = 725.0;
 alpts[i][y] = 265.0;
 i = i+1;
 alpts[i][x] = 800.0;
 alpts[i][y] = 260.0;
 i = i+1;
 alpts[i][x] = 875.0;
 alpts[i][y] = 245.0;
 i = i+1;
 alpts[i][x] = 925.0;
 alpts[i][y] = 230.0;
 i = i+1;
 alpts[i][x] = 1000.0;
 alpts[i][y] = 225.0;
 i = i+1;
 alpts[i][x] = 1050.0;
 alpts[i][y] = 235.0;
 ntot = 16;
break;
}


 /* END defout */
 }
```

PROGRAM : LIGHTDEFS.H

```
/*  This file contains the material, light and lighting model defs
    This is lightdefs.h                                    */

/*  define the appalachian material light */

#define  MATFOREST 1

static float dkgreen[] = {
                          EMISSION,   0.0, 0.0, 0.0,
                          AMBIENT,    0.015, 0.465, 0.025,
                          DIFFUSE,    0.105, 0.545, 0.115,
                          SPECULAR,   0.0, 0.0, 0.0,
                          SHININESS,  0.0,
                          ALPHA,      1.0,
                          LMNULL
                        };

/*  define the granite material light */

#define MATGRANITE 2

static float ltgray[] = {
                          EMISSION,   0.0, 0.0, 0.0,
                          AMBIENT,    0.400, 0.415, 0.505,
                          DIFFUSE,    0.405, 0.405, 0.500,
                          SPECULAR,   0.464, 0.364, 0.564,
                          SHININESS,  0.0,
                          ALPHA,      1.0,
                            LMNULL
                        };

/*  define the alpine material light */

#define MATSUNSET 3

static float mdorange[] = {
                           EMISSION,   0.0, 0.0, 0.0,
                           AMBIENT,    0.665, 0.285, 0.005,
                           DIFFUSE,    0.725, 0.465, 0.235,
                           SPECULAR,   0.0, 0.0, 0.0,
                           SHININESS,  0.0,
                           ALPHA,      1.0,
                           LMNULL
                         };
```

```
/*  define the smooth rolling hills material light */

#define MATGRASS 4

static float ltgreen[] = {
                         EMISSION,   0.0, 0.0, 0.0,
                         AMBIENT,    0.025, 0.905, 0.035,
                         DIFFUSE,    0.245, 0.994, 0.155,
                         SPECULAR,   0.0, 0.0, 0.0,
                         SHININESS,  0.0,
                         ALPHA,      1.0,
                         LMNULL
                       };

/*  define the rocky mountain material light */

#define MATROCKS 5

static float ltbrown[] = {
                          EMISSION,   0.0, 0.0, 0.0,
                          AMBIENT,    0.565, 0.335, 0.035,
                          DIFFUSE,    0.585, 0.415, 0.325,
                          SPECULAR,   0.0, 0.0, 0.0,
                          SHININESS,  0.0,
                          ALPHA,      1.0,
                          LMNULL
                        };

/*  define the light source for morning*/

#define LSOURCM 6

static float lghtsourcm[] = {
                           AMBIENT,   0.3, 0.2, 0.1,
                           LCOLOR,    1.0, 0.8, 0.0,
                           POSITION, -1.0, 1.5, 0.5, 0.0,
                           LMNULL
                         };

/*  define the light source for noon*/

#define LSOURCN 12

static float lghtsourcn[] = {
                           AMBIENT,   0.6, 0.4, 0.4,
                           LCOLOR,    1.0, 0.8, 0.0,
                           POSITION,  0.0, 2.0, 1.0, 0.0,
                           LMNULL
                         };
```

```
/*  define the light source for evening*/

#define LSOURCE 13

static float lghtsource[] = {
                            AMBIENT,  0.4, 0.2, 0.1,
                            LCOLOR,   1.0, 0.8, 0.0,
                            POSITION, 1.5, 1.0, 1.3, 0.0,
                            LMNULL
                        };

/*  define the light model for morning*/
#define LMODM 7

static float lghtmodm[] = {
                           AMBIENT,     0.2, 0.2, 0.2,
                           LOCALVIEWER, 0.0,
                           ATTENUATION, 1.0, 0.0,
                           LMNULL
                       };

/*  define the light model for noon*/
#define LMODN 10

static float lghtmodn[] = {
                           AMBIENT,     0.5, 0.5, 0.5,
                           LOCALVIEWER, 0.0,
                           ATTENUATION, 1.0, 0.0,
                           LMNULL
                       };

/*  define the light model for evening*/
#define LMODE 11

static float lghtmode[] = {
                           AMBIENT,     0.3, 0.2, 0.2,
                           LOCALVIEWER, 0.0,
                           ATTENUATION, 1.0, 0.0,
                           LMNULL
                       };
```

PROGRAM : LIGHTINIT.C

```
/* This file is lightinit.c
   This file initializes the light material, source and scene light
    for ISFM
   All the definitions are in lightdefs.h                          */

#include "gl.h"
#include "lightdefs.h"

/* function to initialize the mountain material   */

   initialmat()
    {
     /* Make defs for the mountain material    */

     lmdef(DEFMATERIAL,MATFOREST,21,dkgreen);
     lmdef(DEFMATERIAL,MATGRANITE,21,ltgray);
     lmdef(DEFMATERIAL,MATSUNSET,21,mdorange);
     lmdef(DEFMATERIAL,MATGRASS,21,ltgreen);
     lmdef(DEFMATERIAL,MATROCKS,21,ltbrown);

    }

/* function to initialize the light source   */

   initialls()
    {
     /*  Make def for the light source  */

     lmdef(DEFLIGHT,LSOURCM,14,lghtsourcm);
     lmdef(DEFLIGHT,LSOURCN,14,lghtsourcn);
     lmdef(DEFLIGHT,LSOURCE,14,lghtsource);

    }

/* function to initialize the scene light  */

   initiallm()
    {
     /* Make def for the scene light   */

     lmdef(DEFLMODEL,LMODM,10,lghtmodm);
     lmdef(DEFLMODEL,LMODN,10,lghtmodn);
     lmdef(DEFLMODEL,LMODE,10,lghtmode);

    }
```

PROGRAM : LIGHTMNT.C

```
/* This routine will pick up the stored values and then
    light the polygons  */

#define x 0
#define y 1
#define z 2
#define abs(x)   (x)>0  ?  (x) : -(x)


#include "gl.h"
#include "device.h"
#include "list.h"
#include "stdio.h"
#include "lightdefs.h"
#include "math.h"

/* The variables are the following:
    IC - counter for current base triangle
    frst,tail - pointers to addresses of the array for
       storing the vertices
    AS[10] - array for storing endpoints of the triangles in
       dynamic storage
                                                          */
extern int IC;
extern LINK frst,tail;
extern float AS[10];


   litemnt()

   {
   /* Local variables are the following:
      i - loop variable
      PL1,PL2,PL3 - temporary values used for lighting the triangles
      ysno - flag to determine if triangle small enough
      NL1, NLS, NS - variables to determine the normals of each
          triangle surface
      */

   int i;
   /*float temp[3];*/
   float PL1[3],PL2[3],PL3[3];
   float ysno;
   float NL1[3],NLS[3],NS;


    IC = IC + 1;
```

```
/* get the values of the endpoints from the dynamic array */
 for(i=0;i<=2;i=i+1)
    {

    PL1[i] = AS[i];
    PL2[i] = AS[i+3];
    PL3[i] = AS[i+6];
    }
    ysno = AS[9];

 /* create the vector normal to the triangle surface       */
  NLS[0]=((PL2[y]-PL1[y])*(PL3[z]-PL1[z]))-((PL3[y]-PL1[y])*(PL2[z]-PL1[z]));
  NLS[1]=((PL2[z]-PL1[z])*(PL3[x]-PL1[x]))-((PL3[z]-PL1[z])*(PL2[x]-PL1[x]));
  NLS[2]=((PL2[x]-PL1[x])*(PL3[y]-PL1[y]))-((PL3[x]-PL1[x])*(PL2[y]-PL1[y]));

  NS = (NLS[0]*NLS[0])+(NLS[1]*NLS[1])+(NLS[2]*NLS[2]);
  NS = sqrt(NS);

  NL1[0] = NLS[0]/NS;
  NL1[1] = NLS[1]/NS;
  NL1[2] = NLS[2]/NS;

      if (ysno >= 1.000000)
      {

     /* light the triangle and send to the screen  */
      bgnpolygon();
      n3f(NL1);
      v3f(PL1);
      v3f(PL2);
      v3f(PL3);
      endpolygon();

      }

   return;

   }
```

PROGRAM : TUTSUB.C

```
/*   subroutine that does the tutorial                    */

#define EXIT 95

#include "gl.h"
#include "device.h"
#include "list.h"
#include "math.h"
#include "stdio.h"

  tutorl()
  {
  /* The local variables are the following:
     tutmenu - integer variable for tutorial menu
     pupval - integer value for pop-up menus
     value - value for reading from the queue
                                                        */
  int tutmenu;
  int pupval;
  short value;

 /* Create the tutorial menu                              */
  tutmenu = defpup("Tutorial %t |On Texture %x11");
  addtopup(tutmenu,"On Lighting Control %x12");
  addtopup(tutmenu,"On Structural Factor %x13");
  addtopup(tutmenu,"On Create a Mountain %x14 ");
  addtopup(tutmenu,"On Outline %x15 ");
  addtopup(tutmenu,"On Create a Mountain Range %x16 ");
  addtopup(tutmenu,"Clear  %x17");
  addtopup(tutmenu,"exit %x95");

 /* clear the screen with a blue background              */
   RGBcolor(0,175,255);
   clear();

 /* set the color to black for the lettering            */
  RGBcolor(0,0,0);

 /* name the title for the window and write out initial introduction
    to the tutorial
                                                        */
  wintitle("Tutorial");
   cmov2i(275,725);
   charstr("Welcome to the Tutorial section for ISFM");
   cmov2i(175,650);
   charstr(" This tutorial section will describe the options and defaults");
   cmov2i(175,625);
   charstr("available in ISFM");
   cmov2i(175,575);
   charstr(" Press Menubutton to see the Contents");
```

```
/* stay in loop until EXIT is chosen                    */
 while(pupval != EXIT)
 {
   switch(qread(&value))
   {

    case REDRAW:
      reshapeviewport();
      break;

    case MENUBUTTON:
      if (value == 1)
      {
        pupval = dopup(tutmenu);
        switch(pupval)
        {
      case 11:                  /* Tutorial on Texture   */
        wintitle("Tutorial on Texture");
        RGBcolor(0,175,255);
        clear();
        RGBcolor(0,0,0);
        cmov2i(275,800);
        charstr("Tutorial on Texture");
        cmov2i(175,725);
        charstr(" The texture of the Mountain is the roughness or smoothness");
        cmov2i(175,700);
        charstr("of the surface. The possible types of mountains  ranging");
        cmov2i(175,675);
        charstr("from the most rugged to the smoothest are: ");
        cmov2i(225,625);
        charstr(" a) rugged granite with a white and gray color ");
        cmov2i(225,600);
        charstr(" b) rocky mountains with a spiked brown color ");
        cmov2i(225,575);
        charstr(" c) alpine mountain with a rust and brown color ");
        cmov2i(225,550);
        charstr(" d) appalachian with a forest green color");
        cmov2i(225,525);
        charstr(" e) smooth rolling hills with a grass green color ");
        cmov2i(150,375);
        charstr(" The default value is an alpine mountain");
        break;

      case 12:                /* Tutorial on Lighting Control  */
        RGBcolor(0,175,255);
        clear();
        RGBcolor(0,0,0);
        cmov2i(275,800);
        charstr("Tutorial on Lighting Control");
        cmov2i(175,725);
        charstr(" There are three positions the sun can be located");
        cmov2i(225,675);
        charstr("a) Morning  The sun is located at an angle 30 degrees");
        cmov2i(250,650);
        charstr("from the horizontal negative x axis and 30 degrees  ");
```

```
    cmov2i(250,625);
    charstr("out from the xy plane  ");
    cmov2i(225,575);
    charstr("b) Noon  The sun is located at an angle 60 degrees ");
    cmov2i(250,550);
    charstr("from the horizontal and directly behind the person.  ");
    cmov2i(225,500);
    charstr("c) Evening  The sun is located at an angle 30 degrees  ");
    cmov2i(250,475);
    charstr("from the horizontal positive x axis and 30 degrees  ");
    cmov2i(250,450);
    charstr("out from the xy plane  ");
    cmov2i(150,375);
    charstr(" The default value is a morning location");
    break;

  case 13:          /* Tutorial on Structural Factor   */
    RGBcolor(0,175,255);
    clear();
    RGBcolor(0,0,0);
    cmov2i(275,800);
    charstr("Tutorial on Structural Factor");
    cmov2i(175,725);
    charstr(" The structural factor determines how detailed the");
    cmov2i(175,700);
    charstr(" surface structure is.  The two options are:");
    cmov2i(225,650);
    charstr("a) Medium  The number of recursion levels stop when ");
    cmov2i(250,625);
    charstr("the distance of all three edges is less than or ");
    cmov2i(250,600);
    charstr("equal to 16 pixels ");
    cmov2i(225,550);
    charstr("a) Fine  The number of recursion levels stop when ");
    cmov2i(250,525);
    charstr("the distance of all three edges is less than or ");
    cmov2i(250,500);
    charstr("equal to 9 pixels ");
    cmov2i(150,425);
    charstr("The default structure factor is medium  ");
    break;

  case 14:         /*  Tutorial on create a Mountain   */
    RGBcolor(0,175,255);
    clear();
    RGBcolor(0,0,0);
    cmov2i(275,800);
    charstr("Tutorial on Create a Mountain");
    cmov2i(175,725);
    charstr(" This option is used to create a single mountain.");
    cmov2i(175,700);
    charstr(" The Create a Mountain option allows one to view the");
    cmov2i(175,675);
    charstr("different available types of fractal mountains quickly");
```

```
    cmov2i(250,625);
    charstr(" To execute the create a mountain there is a submenu ");
    cmov2i(200,600);
    charstr("giving the magnification factor.  The magnification of ");
    cmov2i(200,575);
    charstr("1.0, 2.0, 3.0 and 4.0 allows one to see a closer view ");
    cmov2i(200,550);
    charstr("of what the structure really looks like ");
    cmov2i(150,425);
    charstr("The default magnification is 1.0 ");
    break;
  case 15:                /* Tutorial on Outline        */
    RGBcolor(0,175,255);
    clear();
    RGBcolor(0,0,0);
    cmov2i(275,800);
    charstr("Tutorial on Outline");
    cmov2i(175,725);
    charstr(" The outline section is for creating the contour of the.");
    cmov2i(175,700);
    charstr("mountain range.  The outline options are to either ");
    cmov2i(175,675);
    charstr("create the contour of the mountain range or to clear");
    cmov2i(175,650);
    charstr("and reset the contour already created.");
    cmov2i(200,600);
    charstr(" a) To create the outline the following things can be done ");
    cmov2i(250,575);
    charstr(" 1) press the left mouse button to draw the outline ");
    cmov2i(250,550);
    charstr(" 2) press the middle mouse button for a new location ");
    cmov2i(250,525);
    charstr(" 3) press the right mouse button to erase the points one ");
    cmov2i(240,500);
    charstr(" at a time starting with the last point and working back.");
    cmov2i(250,475);
    charstr(" 4)press the keyboard 'e' value to exit the outline section");
    cmov2i(200,450);
    charstr(" b) Clear and reset- clears the screen display and reset ");
    cmov2i(250,425);
    charstr(" initializes the point array to 0.0, so a new outline can ");
    cmov2i(250,400);
    charstr(" be created. ");
    cmov2i(200,375);
    charstr(" To change the current outline go back into create and the ");
    cmov2i(200,350);
   charstr(" cursor will start at the last point created of the previous");
    cmov2i(200,325);
    charstr(" outline. ");
    cmov2i(150,250);
    charstr("A default contour for each type of mountain is available ");
    break;
```

```
        case 16:            /* Tutorial on create a Mountain Range  */
         RGBcolor(0,175,255);
         clear();
         RGBcolor(0,0,0);
         cmov2i(275,800);
         charstr("Tutorial on Create a Mountain Range");
         cmov2i(175,725);
         charstr(" The create a mountain range option is used to create.");
         cmov2i(175,700);
         charstr(" a whole mountain scene.");
         cmov2i(175,675);
         charstr("If the outline or the other options are not selected the");
         cmov2i(175,650);
         charstr("default options will be used.  ");
         break;
        case 17:                  /* for clear  */
         wintitle("Tutorial");
         RGBcolor(0,175,255);
         clear();
         break;
        }
       }
       break;
      default:
      break;

      }
     }

/* reset window title and clear the screen          */
  wintitle("Fractal Mountain");
  RGBcolor(0,175,255);
  clear();

  return;
 }
```

REFERENCES

1. Mandelbrot, Benoit B., The Fractal Geometry of Nature, W.H. Freeman and Company, 1983

2. Sobelman, Gerald E., and Dredelberg, David E., Advanced C: Techniques and Applications, Que Corporation, 1988

3. La Brecque, Mort, "Fractal Applications", Mosaic, Vol 17, No 4, pp 34-38, Winter 1986/87

4. Goodchild, Michael F., "Fractals and the Accuracy of Geographical Measures", Mathematical Geology, Vol 12, No 2, pp 85-98, 1980

5. Voss, Richard F., "Random Fractal Forgeries", Paper presented at Siggraph 12th course on Fractals: Basic Concepts, Computation and Rendering, San Francisco, California, 23 July 1985

6. Mandelbrot, Benoit B., "Topics on the Midpoint Displacement Technique and Its Application to Model Reliefs and Coastlines", Paper presented at Siggraph 12th course on Fractals: Basic Concepts, Computation and Rendering, San Francisco, California, 23 July 1985

7. Demko, S., Hodges, L., Naylor, B. "Construction of Fractal Objects with Iterated Function Systems", Computer Graphics, Vol 19, No 3, pp 271-278, July 1985

8. Amburn, P., Grant, E., Whitted, T., " Managing Geometric Complexity with Enhanced Procedural Models", Computer Graphics, Vol 20, No 4, pp 189-195, August 1986

9. Miller, Gavin S.P., "The Definition and Rendering of Terrain Maps", Computer Graphics, Vol 20, No 4, pp 39-48, August 1986

10. Carpenter, Loren C., "Additional Perspectives on Fractals", College Mathematics Journal, Vol 15, No 2, p 119, 1984

11. Smith, Alan Ray, "Plants, Fractals and Formal Languages", Computer Graphics, Vol 18 No 3, pp 1-10, July 1984

12. Carpenter, Loren C., "Computer Rendering of Fractal Curves and Surfaces", Paper presented at Siggraph, 13th, Course on Fractals: Basic Concepts, Computation and Selected Topics, Dallas, Texas, 18 August 1986

13. Fournier, A., Fussell, D., Carpenter, L., "Computer Rendering of Stochastic Models", Communications of the ACM, Vol 25, No 6, pp 371-384, June 1982

14. Peitgen, Heinz-Otto, and others, The Science of Fractal Images, Springer-Verlag, 1988

15. Sorensen, Peter R., "Fractals", Byte, September 1984

16. Bass, Patrick, "3-D FRACTALS Three-dimensional ST Landscapes", Antic, The Atari Resource, pp 52-56, 101-106, April 1986

17. van de Panne, Michiel, "3-D Fractals", Creative Computing, pp 78-82, July 1985

18. Pentland, Alex, "Fractal-based Description of Natural Scenes", Proceedings of IEEE Conference on Computing Vision and Patttern Recognition '83, Arlington, Virginia, July 1983

19. Pentland, Alex, "Fractal-based Description of Natural Scenes", IEEE Transaction on Pattern Analysis and Machine Intelligence, Vol PAMI-6, No 6, November 1984

20. Naval Postgraduate School, Monterey, NPS52-86-008 The Fractal Geometry of Nature: Its Mathematical Basis and Application to Computer Graphics, by Michael E. Gaddis and Michael J. Zyda, January 1986

21. Sedgewick, Robert, Algorithms, Addison-Wesley Publishing Company, 1983

22. Foley, J. D., Van Dam A., Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Company, 1984

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Professor Michael J. Zyda Naval Postgraduate School Computer Science Department, Code 52Zk Monterey, California 93943	1
4.	Professor Harold M. Fredricksen Naval Postgraduate School Mathematics Department, Code 53Fs Monterey, California 93943	1
5.	Patricia J. Hunt 317 Willow Bend Court Chesapeake, Virginia 23323	2

Thesis
C6155 Collins
c.1 Three-dimensional
fractal mountains.